1-1-2005

Sending non-contiguous data in MPI programs

Yanmei Wang
Iowa State University

Sending non-contiguous data in MPI programs

by

Yanmei Wang

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Glenn R. Luecke, Co-major Professor
Gurpur M. Prabhu, Co-major Professor
Lu Ruan

Iowa State University

Ames, Iowa

2005

Graduate College
Iowa State University

This is to certify that the master's thesis of

Yanmei Wang

has met the thesis requirements of Iowa State University

Signatures have been redacted for privacy

Table of Contents

List of Figures

List of Tables

Abstract

The purpose of this paper is to evaluate the performance and ease-of-use of four methods of sending non-contiguous data in MPI programs. The methods considered in this paper are: (1) using Fortran 90 array sections, (2) using MPI derived types, (3) using explicit user packing into a contiguous buffer, and (4) using explicit packing with mpi_pack and mpi_unpack into a contiguous buffer. Four communication tests, commonly found in scientific applications, were designed and run with a variety of message sizes on a Cray X1, a Cray XT3, an IBM Power4 system, and on an Intel/Myrinet cluster. Methods (1) and (2) were much easier to use than the other methods. Performance of MPI derived types depended on the quality of the implementation and provided the best performance compared with the other methods on the IBM and Cray XT3 machines.

Chapter 1. General Introduction

Introduction

The Message Passing Interface (MPI) standard was introduced in 1994. MPI is a message-passing library, a collection of routines that enable passing messages in Fortran, C and C++ among the processors in a distributed memory parallel computer. MPI derived datatypes provide a convenient way to send non-contiguous data in a single communication. Non-contiguous data can also be sent by explicitly copying the data into a contiguous buffer and sending (receiving) the contiguous buffer. A third method of sending (receiving) non-contiguous data is using mpi_pack (and mpi_unpack) to copy the data into a contiguous buffer for sending (receiving). When non-contiguous data can be represented by a Fortran 90 array section, then this data can be sent directly; for example, call mpi_send(A(1:5:2), 3, mpi_real, ...). The purpose of this study is to evaluate the performance and ease-of-use of these methods for sending non-contiguous data for a variety of constructs commonly found in scientific applications.

Four communication tests were chosen to represent commonly-used scientific operations involving sending noncontiguous data: sending row blocks of 2-dimensional arrays, sending elements with uniform stride in 1-dimensional arrays, sending the lower triangular portion of 2-dimensional arrays, and sending block diagonals of 2-dimensional arrays. These tests were run on the Cray X1, the Cray XT3, the IBM DataStar, and on an Intel/Myrinet cluster.

Thesis Organization

In Chapter 2, paper "sending non-continguous data in MPI programs" is presented. Yanmei Wang is the primary researcher and author of this paper. This paper evaluated the performance and ease-of-use of four methods for sending non-contiguous data in MPI programs. Four designed test run on the Cray X1, the Cray XT3, the IBM DataStar. A general conclusion is given in chapter 3.

Chapter 2. Sending Non-Contiguous Data in MPI Programs

A paper to be submitted to
The Journal of Performance Evaluation and Modelling for Computer Systems

Glenn R. Luecke, Yanmei Wang
Iowa State University
grl@iastate.edu, yanmei@iastate.edu

Abstract

The purpose of this paper is to evaluate the performance and ease-of-use of four methods of sending non-contiguous data in MPI programs. The methods considered in this paper are: (1) using Fortran 90 array sections, (2) using MPI derived types, (3) using explicit user packing into a contiguous buffer, and (4) using explicit packing with mpi_pack and mpi_unpack into a contiguous buffer. Four communication tests, commonly found in scientific applications, were designed and run with a variety of message sizes on a Cray X1, a Cray XT3, an IBM Power4 system, and on an Intel/Myrinet cluster. Methods (1) and (2) were much easier to use than the other methods. Performance of MPI derived types depended on the quality of the implementation and provided the best performance compared with the other methods on the IBM and Cray XT3 machines.

1. Introduction

The Message Passing Interface (MPI) standard was introduced in 1994. MPI is a message-passing library, a collection of routines that enable passing messages in Fortran, C and C++ among processors for distributed memory parallel computers. MPI derived datatypes provide a convenient way to send non-contiguous data in a single communication. Non-contiguous data can also be sent by explicitly copying the data into a contiguous buffer and sending the contiguous buffer. Another method of sending non-contiguous data is using mpi_pack to copy the data into a contiguous buffer for sending. When non-contiguous data can be represented by a Fortran 90 array section, this data can be sent using the array section; for example, call mpi_send(A(1:5:2), 3, mpi_real, ...). The purpose of this paper is to evaluate the performance and ease-of-use of these methods for sending non-contiguous data for a variety of constructs commonly found in scientific applications.

Four communication tests were chosen to represent commonly-used scientific operations involving sending noncontiguous data:

1. sending row blocks of 2-dimensional arrays,
2. sending elements with uniform stride in 1-dimensional arrays,
3. sending the lower triangular portion of 2-dimensional arrays, and
4. sending block diagonals of 2-dimensional arrays.

These tests were run on the Cray X1, the Cray XT3, the IBM DataStar, and on an Intel/Myrinet cluster. All tests were executed on nodes with no other jobs running.

Measuring the performance of MPI derived types is also being done at the University of Karlsruhe in Germany, where they have added MPI derived type performance tests to their SKaMPI (Special Karlsruher MPI) MPI benchmark [7][8]. The SKaMPI MPI benchmarks for MPI derived types do not employ cache flushing techniques whereas the performance measurements in this paper measure memory resident (and not in cache) data. This paper also compares the performance of several of different methods for sending non-contiguous data, whereas the SKaMPI tests do not. The Pallas MPI Benchmarks are now called the Intel MPI Benchmarks [12] since Intel purchased Pallas. However, these tests do not include evaluating the performance of MPI derived types.

The Cray X1 is a nonuniform memory access (NUMA) machine consisting of multiple node modules. Each node module contains four multistreaming processors (MSPs), along with either 16 or 32 GB of flat, shared memory plus hardware to support high-speed node to node communication. For more information about the Cray X1, see [1]. All tests have been compiled to MSP mode using Cray Fortran compiler, version 5.4.0.0.10. The version of MPI is mpt.2.4.0.2. This version of MPI on the Cray X1 is based on MPICH1 from Argonne National Laboratory.

The Cray XT3 used was 151 nodes. Each node is comprised of single processor AMD Opteron processor. The communication network to connect nodes is a 3D toroidal grid built by Cray. The system uses Linux on the service nodes and Catamount on the computer nodes,

and uses PGI compilers. The MPI implementation is based on MPICH2 from Argonne National Laboratory. For more information about the Cray XT3, see [2].

The IBM Power4 system used was a 1408 processors machine located in San Diego Supercomputer Center and named DataStar. DataStar has a mix of 176 8-way nodes with 16 GB memory, six 32-way nodes with 128 GB memory and one 32-way node with 256 GB memory. Each Power4 CPU runs at 1.6 GHz. Each Power4 CPU has a two-way associative L1 (32 KB) cache, and a four-way associative L2 (1.4 MB) cache, and the CPU's on a node share an 8-way associative L3 cache (128 MB). All tests are executed on dedicated 8-way nodes. For more information on this machine, see [3].

The Intel/Myrinet cluster used was a 44 dual processor Intel 2.8 GHz Xeon/Myrinet cluster, located at Iowa State University, see [4]. The system was running RedHat 8.0 with SMP enabled (which uses the 2.4.18-14smp Linux kernel) and all tests were used the version 7.1 Intel's Fortran 95 compiler. This machine is running Myrinet's MPI GM libraries based on MPICH version 1.2.5. Myricom [13] does not currently support MPICh2, but they plan to support MPICH2 with the next release of MPICH-MX.

Section 2 introduces the timing methodology employed and section 3 presents each of the tests and performance results. The conclusions are discussed in section 4.

2. Timing Methodology

2.1 Measuring Times

This section describes the timing methodology used for this paper. Round trip ping pong times were measured and then divided by two to obtain the time of sending and receiving a message. Timings can vary significantly if messages are cache resident or memory resident (and not resident in any data cache). Figure 2.1 shows the difference in timings when messages are cache resident and when messages are memory resident on the IBM DataStar for the MPI derived type in test 3 with n = 32. Notice that cache resident message times are

about three times faster than memory resident times on this machine. In this study timings were done with memory resident messages so (data) caches were flushed prior to each timing.

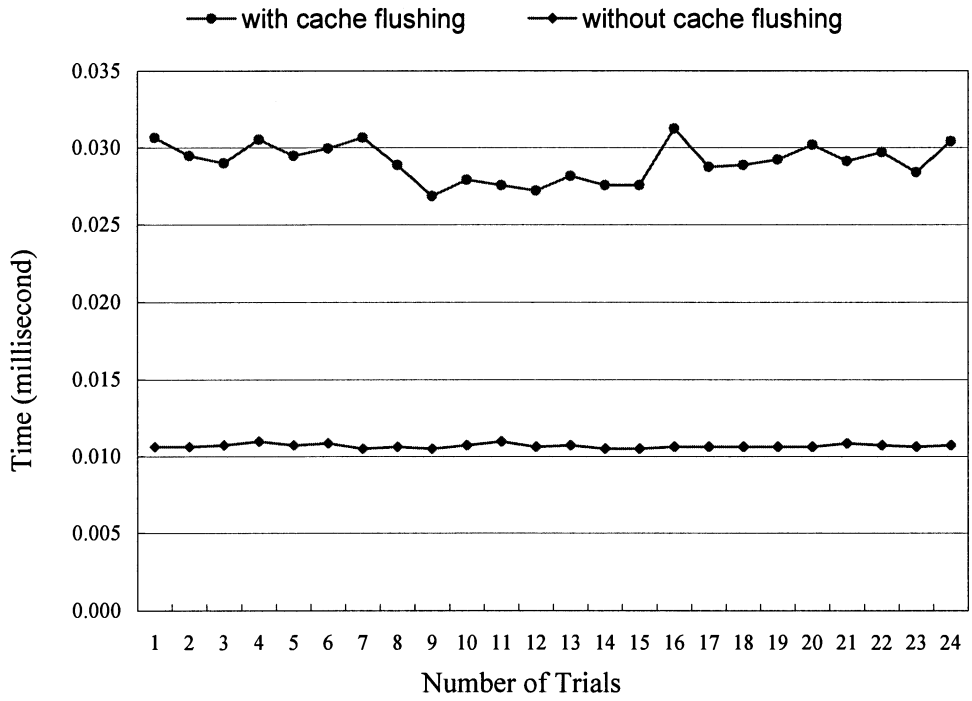

Figure 2.1 Memory resident versus cache resident message comparison for MPI derived type in test 3 within a node on the IBM DataStar .

Most of today's computers are a collection of shared memory nodes interconnected with a communication network for MPI communication. In general, the performance of MPI communication between nodes will be different from communication within a single node. Therefore, timings were performed using p MPI processes and measuring times between MPI process of rank 0 and rank p-1, where p is chosen so the communication will be within a node or between nodes.

The following shows how timings were performed where the k-loop was only executed on the rank 0 and p-1 MPI processes:

```
integer,parameter :: ncache = ..    ! number of 8 byte words in the highest level cache
double precision :: flush(ncache), x
```

```
integer,parameter :: ntrial=51        ! number of timing trials
double precision(:,:) :: ary_time
x = 0.d0
call random_number(flush)
call mpi_type_vector ( ... )      ! define MPI derived type
call mpi_type_commit ( ... )
  .....
 do k = 1, ntrial
      flush(1:ncache) = flush(1:ncache)  + x    ! flush the cache
       call mpi_barrier(mpi_comm_world, ierror)
       if (rank == 0) then
           t = mpi_wtime()    ! time in seconds
           call mpi_send (A... )
           call mpi_recv (B... )
           ary_time(k,j) = 0.5*(mpi_wtime() – t)   ! measure time & divide by 2
          ! The following lines are for preventing compile optimization
           i = min(k, j,  ncache)
           A(i,i) = A(i,i) + 0.01d0*(B(i,i) + flush(i))
           x = x + A(i,i)*0.1d0
      elseif (rank == p-1) then
           call mpi_recv ( A... )
           call mpi_send (B ...)
       endif
       call mpi_barrier(mpi_comm_world, ierror)
   enddo
print *, flush(1),+A(1,1)+B(1,1)   ! prevent dead code elimination by the compiler
```

The flush array was chosen large enough to flush all (data) caches and was set to different sizes depending on the machine used. Ping pong timings were performed using two distinct buffers, A and B. This was needed to ensure that buffers were memory resident. For example, when process j receives data in A, then all or a part of A will be in cache. If A is then sent back to processor of rank 0, then the timings will be faster since A is (partially) cache resident. Thus, the message is sent back using B and is received in B since B is not cache resident on either MPI processes.

The first call to mpi_barrier guarantees that all processes reach this point before calling mpi_wtime. The second call to mpi_barrier is to ensure that no process starts the next trial until all processes have completed timing the ping pong operation.

Most compilers perform optimizations that might change the program, e.g. loop splitting, dead code elimination, prefetching of data. These optimizations may affect the accuracy of the measured ping pong times. All tests were compiled with the –O0 compiler option that is supposed to turn off optimization. However, the above program was carefully written to ensure accurate timings even if compiler optimizations are performed. (We did try running some of our tests on the Intel/Myrinet cluster using Intel's Fortran compiler with the –O0 and –O options and no performance differences were found.)

2.2 Variability of Timings within Nodes

It is important to perform multiple timings for each test to determine the variability of the timings. The smaller the messages being sent, the more variability there will be in the measured times, so we chose the smallest message size in test 1 with MPI derived data types in section 3 to study the variability of the measured times. We set our timing program to time 500 ping pongs and then ran this program twice on a single 2 processor node of the Intel Xeon/Myrinet cluster. The nodes on this machine are dedicated to running only our MPI program. Figure 2.2 shows the results of these two runs. Notice the shifts in average times both within a single run and between multiple runs. These shifts in timing are likely due to the starting and stopping of various processes executing under the control of the operating system. Notice that there are two MPI processes using both of the two physical processors on the same node. Therefore, the operating system processes must share the two processors with the MPI processes and hence interfere with the execution of the MPI program. This program was run at different times during the day and on different days. The average varied

from 0.24 to 0.28 milliseconds yielding a maximum variation of about 17%.

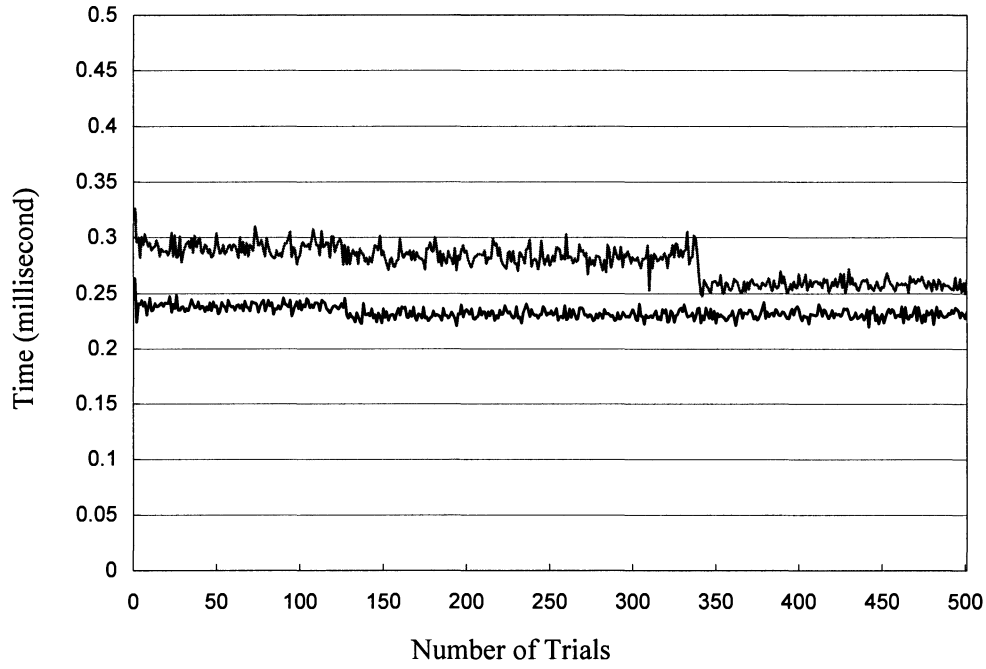

Figure 2.2 Within node timing results for two runs of test 1 with the MPI derived type method on the Intel/Myrinet cluster.

The timing results when running this same program on the IBM DataStar machine within a node were more stable than on the Intel/Myrinet cluster. Table 2.1 shows the results of 3 different runs and Figure 2.3 shows the graph of the first run listed in Table 2.1. The maximum variation of times within a node on the IBM machine was less than 5%. This stability of time measurements is likely due to the fact that the ping pong test only used 2 of the 8 processors on the node. Tasks being run by the operating system could then run on processors not involved in the ping pong. Recall that the coefficient of variance is defined to be the standard deviation divided by the average.

Table 2.1 Within a node timing results for test 1 with the MPI derived type method on the IBM DataStar.

Run	Average Time (Millisecond)	Minimum Time (Millisecond)	Maximum Time (Millisecond)	Standard Deviation	Coefficient of Variance
1	2.97E-01	2.89E-01	3.18E-01	5.62E-03	1.89E-02
2	3.04E-01	2.99E-01	3.56E-01	6.41E-03	2.11E-02
3	3.06E-01	3.01E-01	3.66E-01	7.69E-03	2.51E-02

Variability of timing results for the two Cray machines for this same program both within a node and between nodes were similar to variability of results on the IBM machine and were less than 5%

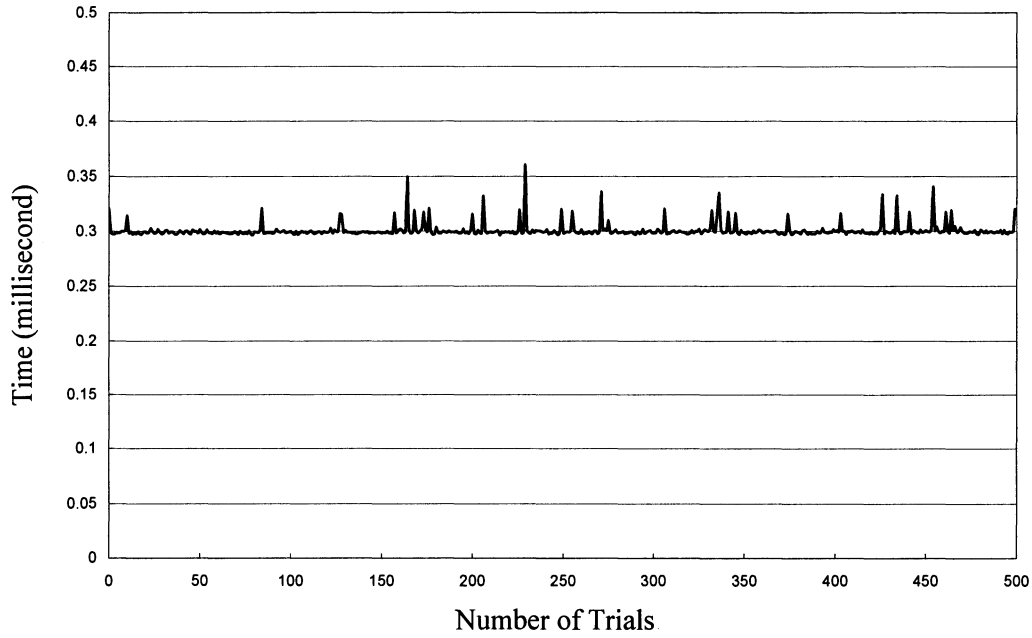

Figure 2.3 Within node timing results for test 1 with the MPI derived type method on the IBM DataStar.

2.3 Variability of Timings between Nodes

When running this same program used in section 2.2 between nodes on the Intel/Myrinet cluster, the timing data was much more stable. Figure 2.4 shows the timing results and Table 2.2 shows 3 timing runs. The data between nodes for this machine varied less than 5%.

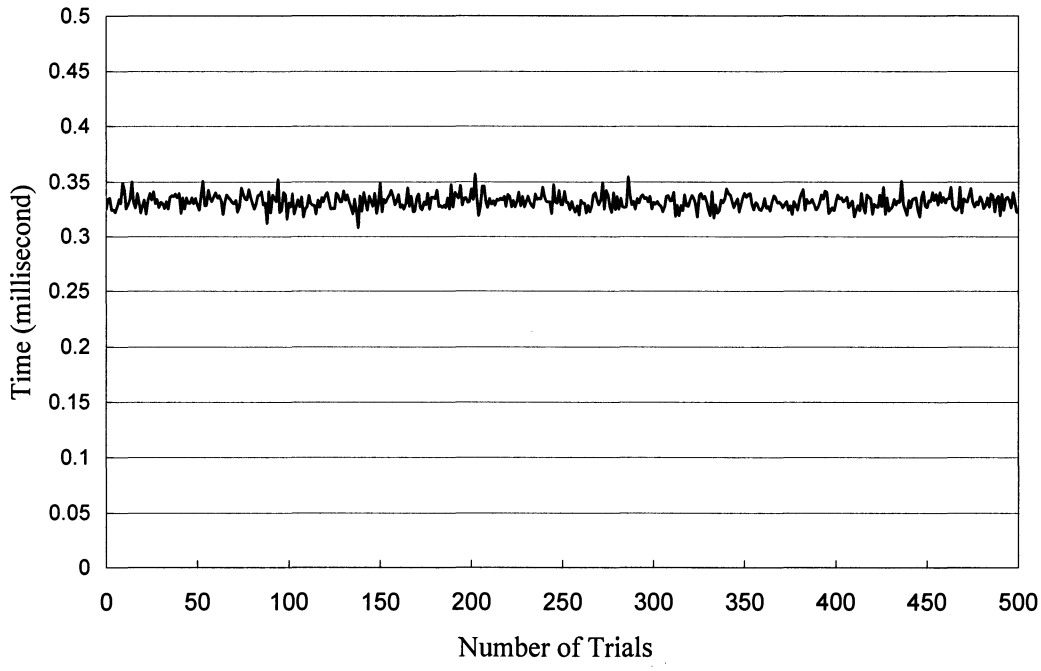

Figure 2.4 Between nodes timing results for test 1 with the MPI derived type method on the Intel/Myrinet cluster.

Table 2.2 Between node timing results for test 1 with the MPI derived type method on the Intel/Myrinet cluster.

Run	Average Time (Millisecond)	Minimum Time (Millisecond)	Maximum Time (Millisecond)	Standard Deviation	Coefficient of Variance
1	3.31E-01	3.13E-01	3.77E-01	7.15E-03	2.16E-02
2	3.28E-01	3.12E-01	3.77E-01	7.67E-03	2.34E-02
3	3.24E-01	3.04E-01	4.16E-01	7.26E-03	2.24E-02

When running this same program between nodes on the IBM DataStar, the timing data was as stable as within a node. Figure 2.5 shows the timing results and Table 2.3 shows 3 timing runs. The data between nodes for this varied at most 5%.

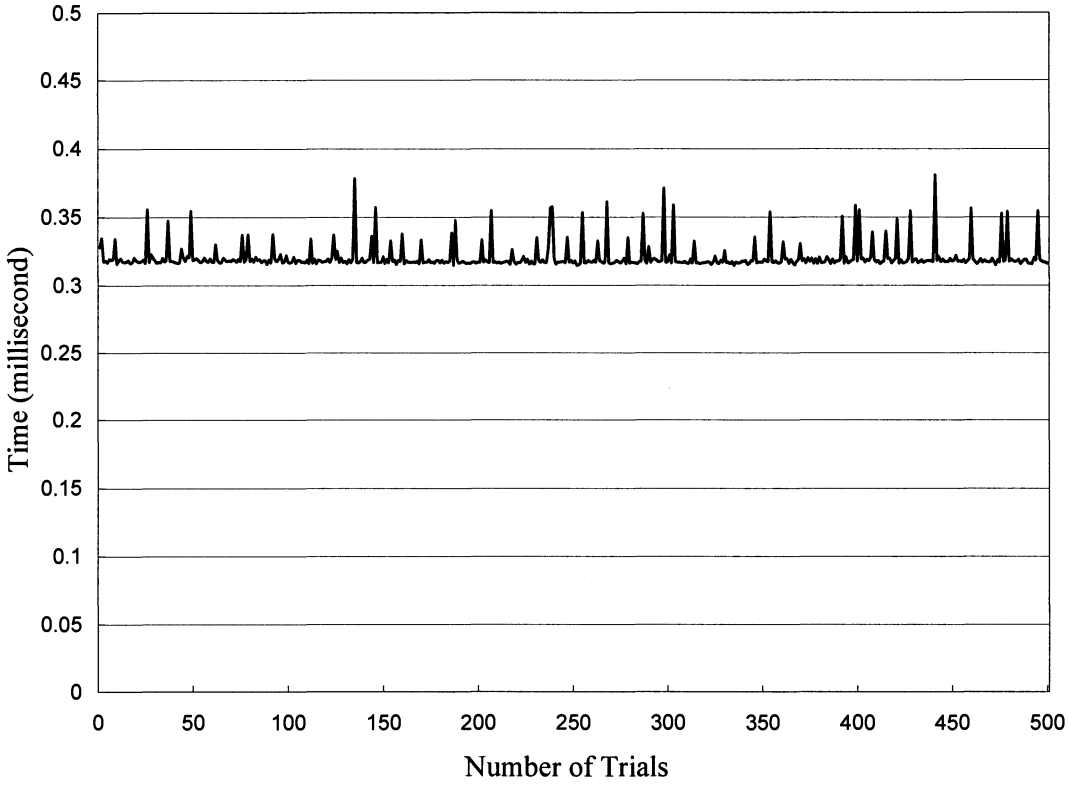

Figure 2.5 Between nodes timing results for test 1 with the MPI derived type method on the IBM DataStar.

Table 2.3 Between nodes timing results for test 1 with the MPI derived type method on the IBM DataStar.

Run	Average Time (Millisecond)	Minimum Time (Millisecond)	Maximum Time (Millisecond)	Standard Deviation	Coefficient of Variance
1	3.20E-01	3.14E-01	3.97E-01	1.02E-02	3.19E-02
2	3.20E-01	3.145E-01	3.81E-01	9.49E-03	2.96E-02
3	3.19E-01	3.14E-01	3.73E-01	7.75E-03	2.43E-02

2.4 Number of Timing Trials

The first time a function/subroutine is called requires additional time that subsequent calls due to the time required for initial set up. Because of this, the first timing trial was always longer than (most) subsequent timing. For this reason, we always discarded the first timing trial.

How many timing trials should one use to compute an average value for the operation being timed? If there are shifts in average times as shown in Figure 2.2, then the average value computed will depend how many timing trials are near to each of the two different average values. In such situations, it is impossible to determine an appropriate number of timing trials to use. Fortunately, most all of the timing trials for all machines and for all tests looked similar to the timing trials shown in Figure 2.6.

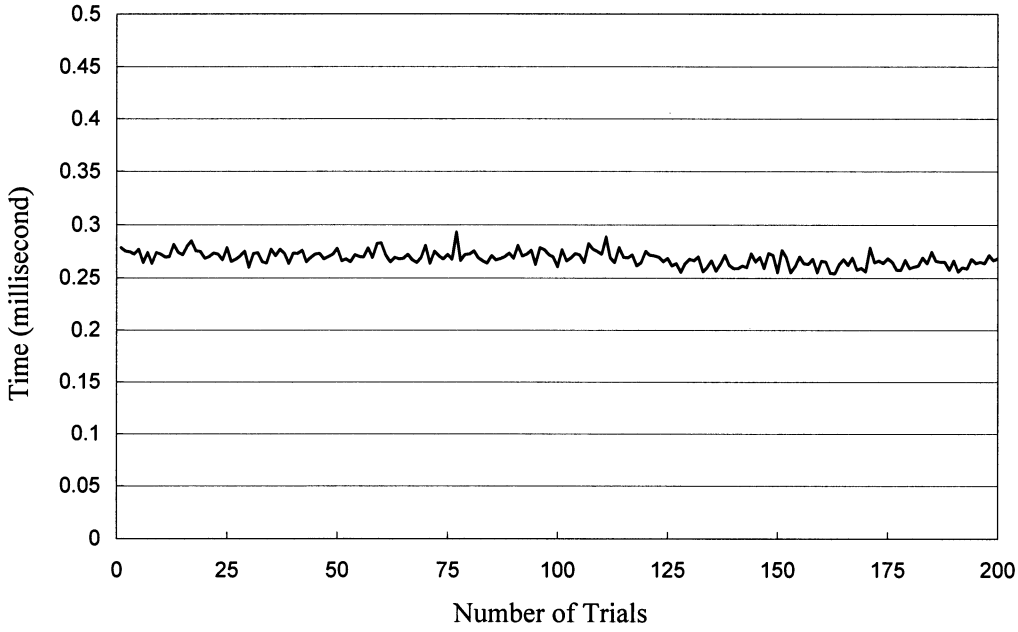

Figure 2.6 Within node timing results for test 1 with the MPI derived type method on the Intel/Myrinet cluster.

To determine how many trials should be used for running all our tests, we took this data and plotted the average times over the first N trials in Figure 2.7. Thus, Figure 2.7 is a graph of average(n) for n = 1, 2, 3, …., 200; where

$$average(n) = (average(1) + average(2) + \ldots + average(n))/n$$

Notice that there is little difference in the average values from roughly 20 trials to 200 trials. We purposely chose the scale for the y-axis in Figure 2.7 to be the same as the scale for all the graphs in this section so it would be easy to compare these average times with the data presented in the other Figures. To be conservative, we always measured 51 times, discarded the first timing, and then took an average of the remaining 50 timing trials.

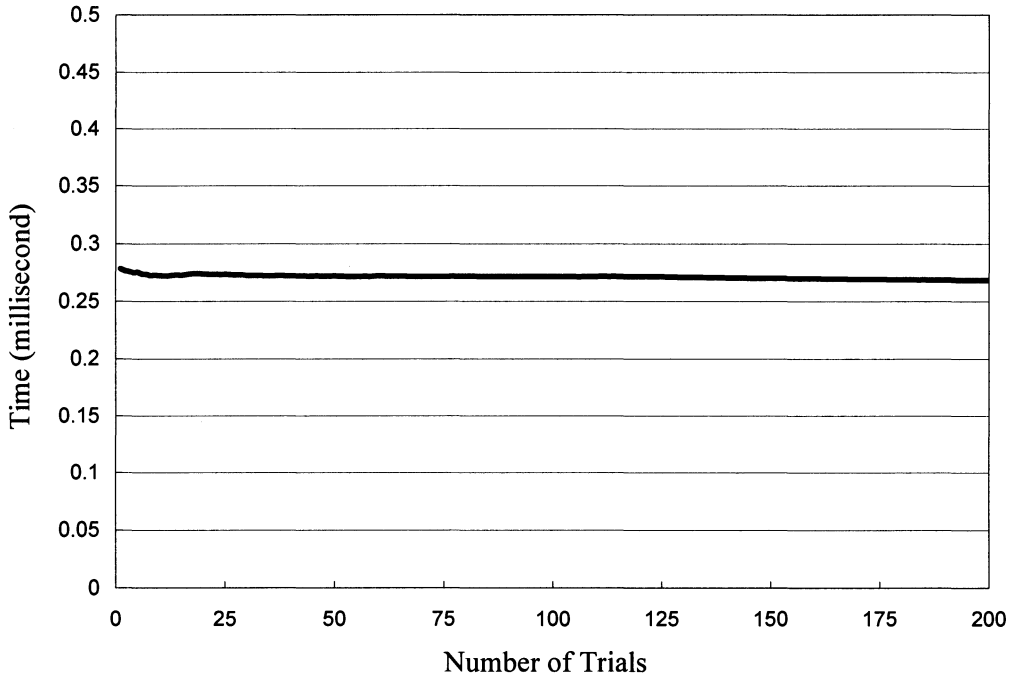

Figure 2.7 Average time for N trials for N = 1, 2, . . . , 200 for test 1 with the MPI derived type method on the Intel/Myrinet cluster within a node.

3. Description of Tests and Performance Results

This section describes each of the five ping pong tests that compare the performance of MPI derived types with using (1) explicit packing, (2) mpi_pack and mpi_unpack, and with using (3) Fortran 90 array sections, when possible. Performance results are presented for different message sizes between nodes and within a shared memory node for the machines listed in section 1. Throughout this section dp denotes mpi_double_precision and comm. denotes mpi_comm_world. The raw data used for computing the ratios for all figures in this section are presented in the appendix.

3.1 Test 1: Sending row blocks of 2-dimensional arrays

Since Fortran stores 2-dimensional arrays by column, sending row blocks of length k of an array A(m,n) involves the sending of noncontiguous data. These row blocks can be sent using any of the four methods described above. To compare their performance, we chose A to be of type double precision and of size 500 by 1000 with row blocks of size 1, 10, 50, 100, 300, 400, 499, and 500. 499 was chosen find out if there was a significant performance difference when sending the entire array with sending all but one row of the array. The size of 500 was used to compare performance with sending contiguous data. We chose the size of A to be large enough so timings would involve hundreds of clock ticks of the timer, mpi_wtime. For all machines, the value of mpi_wtick was about one microsecond. Of course, there are many other sizes of A and row block sizes that could have been used.

The MPI derived type for sending k rows of A was called rowblock and was created as follows:

```
call mpi_type_vector (n, k, m, dp, rowblock, ierror)
call mpi_type_commit (rowblock, ierror)
```

Using the notation of the timing template shown in section 2, the timing for this method can be described as follows:

```
if (rank == 0) then
   t = mpi_wtime ( )
   call mpi_send (A(1, 1), 1, rowblock, ...)
```

```
        call mpi_recv (B(1,1), 1, rowblock, ...)
        time(ktrial, j) = 0.5d0 * (mpi_time( ) – t)
    elseif (rank == j) then
        call mpi_recv (A(1,1), 1, rowblock, ...)
        call mpi_send (B(1,1), 1, rowblock, ...)
    endif
```

The mpi_pack and mpi_unpack routines can also be used to send k rows of A. The sender packs the data to a contiguous buffer, called temp1, using mpi_pack and the receiver unpacks the data using mpi_unpack. This was done as follows, where the integer variable size is set by calling mpi_pack_size(k, dp, comm., size, ierror):

```
if (rank == 0) then
    t = mpi_wtime()
    position = 0
    do i = 1, n
        call mpi_pack(A(1,i), k, dp, temp1, n*size, position, comm, ierror)
    enddo
    call mpi_send(temp1, position, mpi_packed, ...)
    call mpi_recv(temp2, n*size, mpi_packed, ...)
    do i = 1, n
        call mpi_unpack(temp2, n*size, position, B(1,i),k,dp, comm, ierror)
    enddo
    time(ktrial, j) = 0.5d0*(mpi_wtime ( ) – t)
elseif (rank == j) then
    position = 0
    call mpi_recv(temp1, n*size, mpi_packed, ...)
    do i = 1, n
        call mpi_unpack(temp1, n*size, position, A(1,i),k,dp, comm, ierror)
    enddo
    position = 0
    do i = 1, n
       call mpi_pack(B(1,i), k, dp, temp2, n*size,  position, comm, ierror)
    enddo
    call mpi_send(temp2, position, mpi_packed, ...)
endif
```

The k rows of A can also be sent with the user packing the data into a contiguous temporary buffer of size k by n as shown below. Notice the copies are written to insure stride one memory accesses.

```
if (rank == 0) then
   t = mpi_wtime()
   do i = 1, n
      temp1(1:k, i) = A(1:k, i)
    enddo
    call mpi_send(temp1, k*n, ...)
    call mpi_recv(temp2, k*n, ...)
    do i =  1, n
       B(1:k, i) = temp2(1:k, i)
    enddo
    time(ktrial, j) = 0.5d0*(mpi_wtime( ) – t)
elseif (rank == j) then
    call mpi_recv(temp1, k*n, ...)
    do i = 1, n
      A(1:k, i) = temp1(1:k, i)
    enddo
     do i = 1, n
        temp2(1:k, i) = B(1:k, i)
     enddo
     call mpi_send(temp2, k*n, ...)
endif
```

Fortran 90 array sections may also be used to send k rows of A:

```
if (rank == 0) then
    t = mpi_wtime()
    call mpi_send(A(1:k,1:n), k*n, ...)
    call mpi_recv(B(1:k,1:n), k*n, ...)
    time(ktrial, j) = 0.5d0*(mpi_wtime( ) – t)
 elseif (rank == j) then
    call mpi_recv(A(1:k,1:n), k*n, ...)
    call mpi_send(B(1:k,1:n), k*n, ...)
endif
```

Performance results for test 1 are presented in Figures 3.1 through 3.7. For each machine, the performance ratios within a node and between nodes are similar. For the Intel/Myrinet cluster the mpi_pack method performed best for k = 50, 100, 400, and 499 but other methods performed best for the other values of k. For IBM DataStar, MPI derived types performed best for all values of k. For the Cray X1, for k = 1, 10, 50, and 100 Fortran array sections performed best but MPI derived types performed best for the other values of k. For the Cray XT3, MPI derived types performed best for all values of k. Notice that using Fortran array

sections provides nearly the same performance as user packing for all machines and using MPI derived types provides best performance for all machines for most values of k except for the Intel/Myrinet cluster.

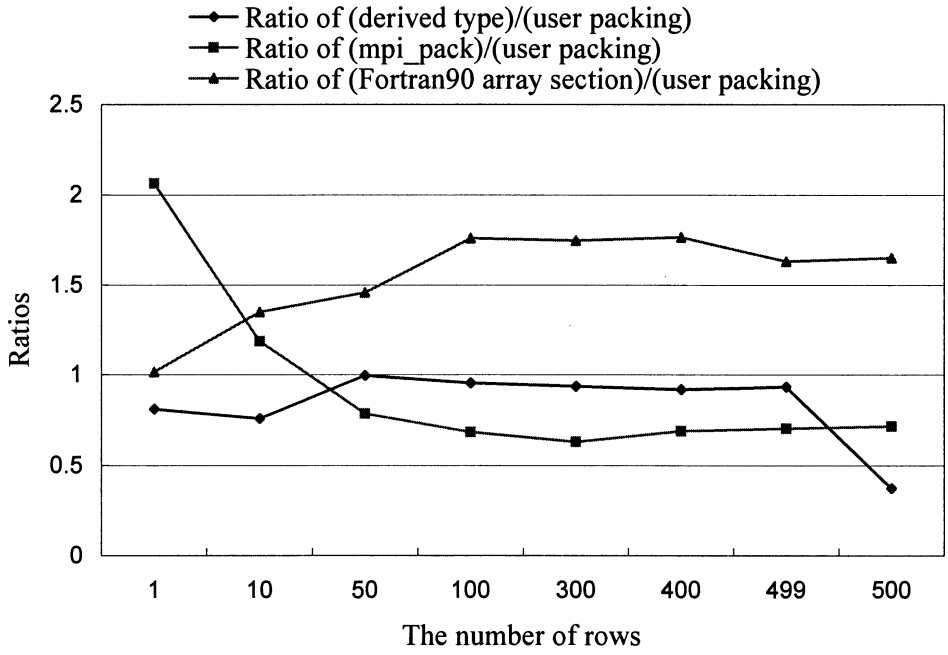

Figure 3.1 Test 1 ratios for the Intel/Myrinet cluster within a node

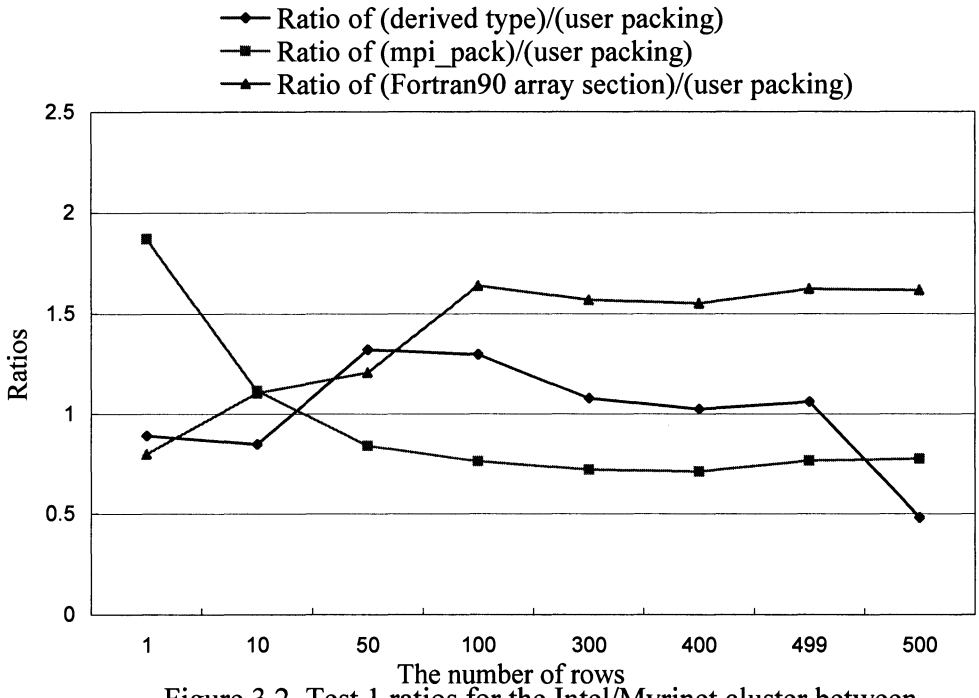

Figure 3.2 Test 1 ratios for the Intel/Myrinet cluster between nodes

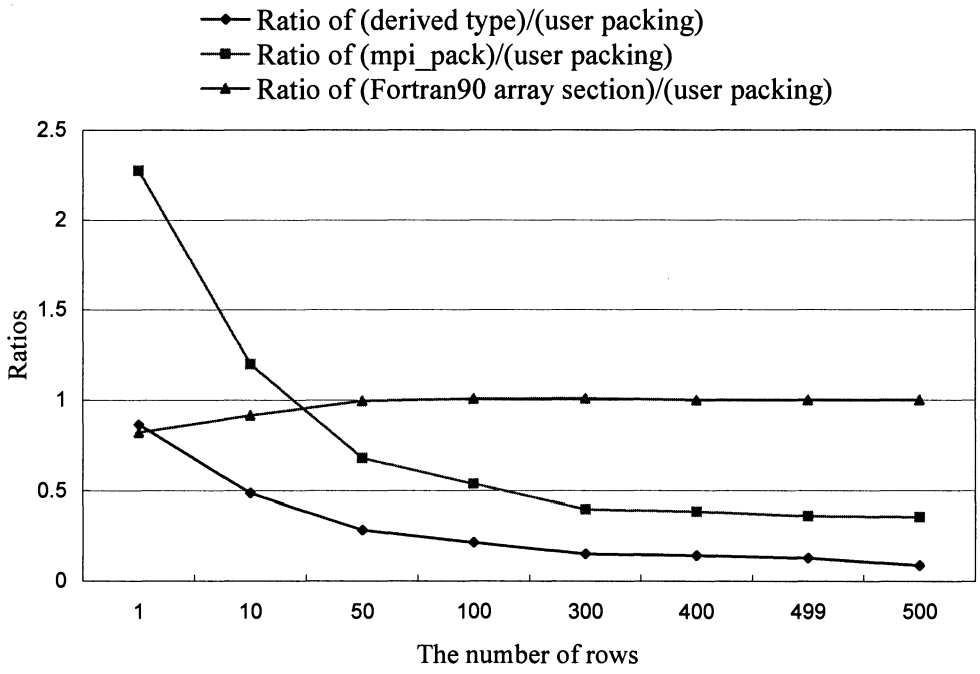

Figure 3.3 Test 1 ratios for the IBM DataStar within a node

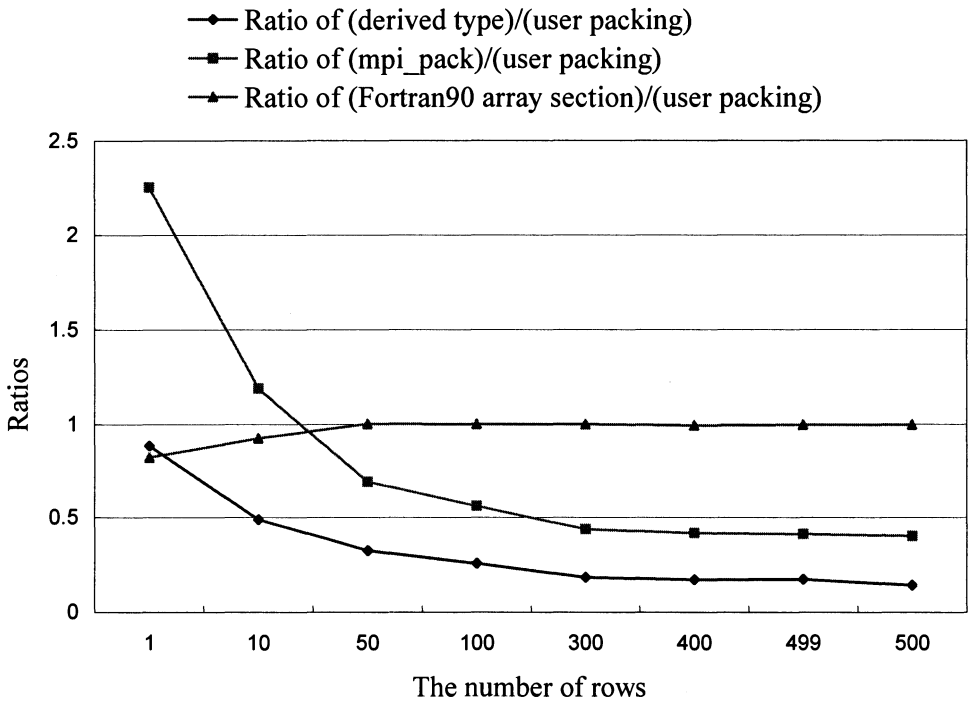

Figure 3.4 Test 1 ratios for the IBM DataStar between nodes

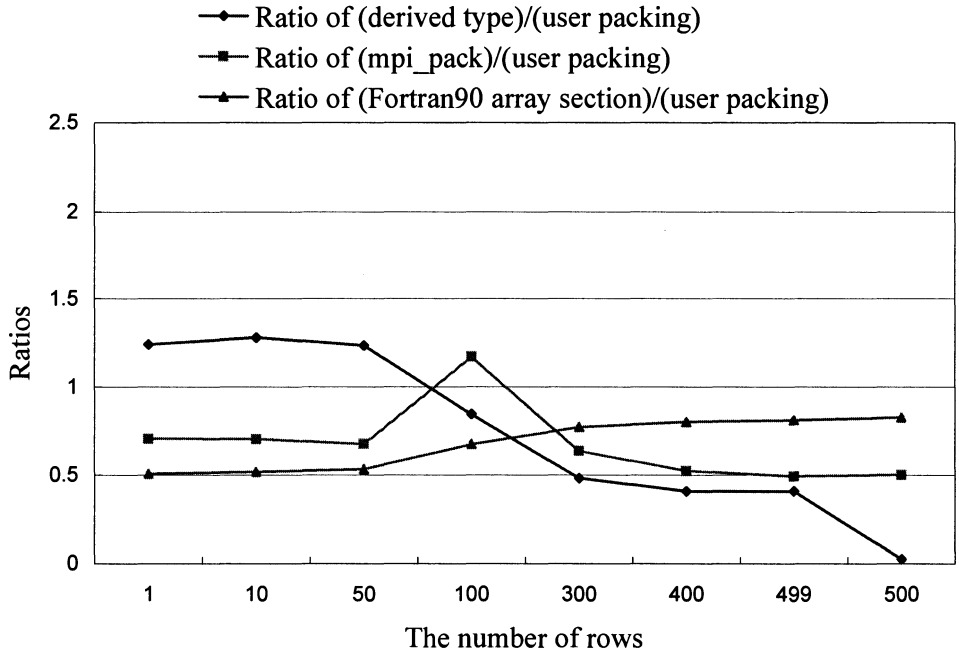

Figure 3.5 Test 1 ratios for the Cray X1 within a node

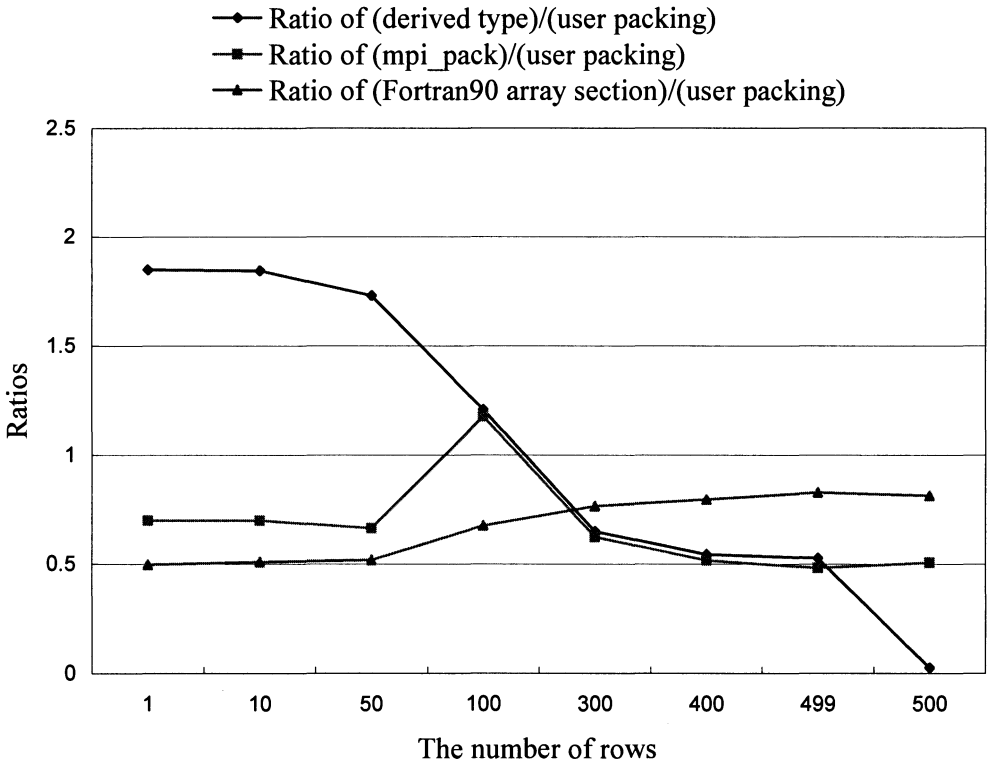

Figure 3.6 Test 1 ratios for the Cray X1 between nodes

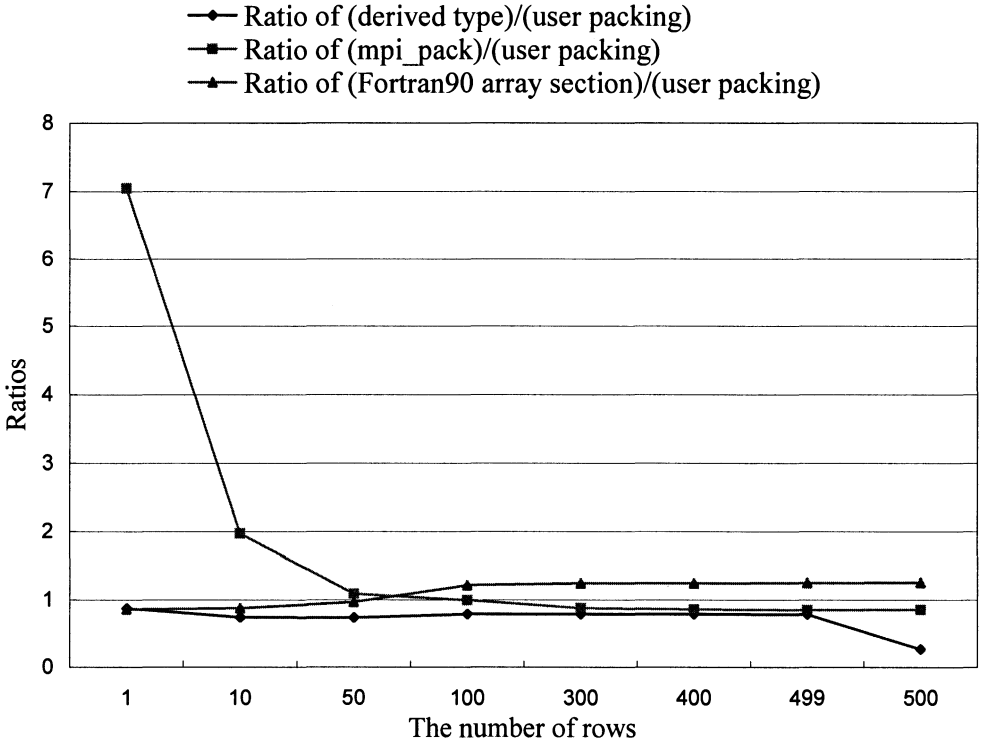

Figure 3.7 Test 1 ratios for the Cray XT3 between nodes

3.2 Test 2. Sending elements with uniform stride in a 1-dimensional array

Sometimes one would like to send data with a uniform stride in a 1-dimensional array. When the stride is greater than one, this means sending data that is not contiguous in memory. Let A and B be double precision 1-dimensional arrays of length n = 5000 and take stride = 2, 3, 4, 5, ..., 17. This noncontiguous data can be sent using all of the methods described in section 3.1. Their timing programs are also similar, so their descriptions are not repeated in this section.

The MPI derived type used to send this data is called data and is defined as follows:

```
count = ceiling(dble(n)/stride)      ! number of elements sent
call mpi_type_vector(count, 1, stride, dp, data, ierror)
call mpi_type_commit(stride, ierror)
```

For the mpi_pack method, the data was packed into temp1 as follows:

```
do i = 0, count-1
   call mpi_pack(A(stride*i+1), 1, dp, temp1, count*size,  &
                        position, comm, ierror)
enddo
```

The user packing method uses:

```
temp1(1:count) = A(1:n:stride)
call mpi_send(temp1, count, dp, ....)
```

The Fortran array section method uses:

```
call mpi_send(A(1:n:stride), count, dp, ....)
```

Performance results for test 2 are presented in Figures 3.8 through 3.14 and include stride one results. For each machine, the performance ratios within a node and between nodes are similar. The performance rations for the Intel/Myrinet cluster, the IBM DataStar, and the Cray XT3 are similar with the mpi_pack method performing poorly and the other methods performing about the same. For the Cray X1, the MPI derived type method performed poorly and the mpi_pack method also performing better but still performing poorly compared

with the other methods. For all machines, excluding the Cray X1, user packing, Fortran array sections, and MPI derived types perform about the same.

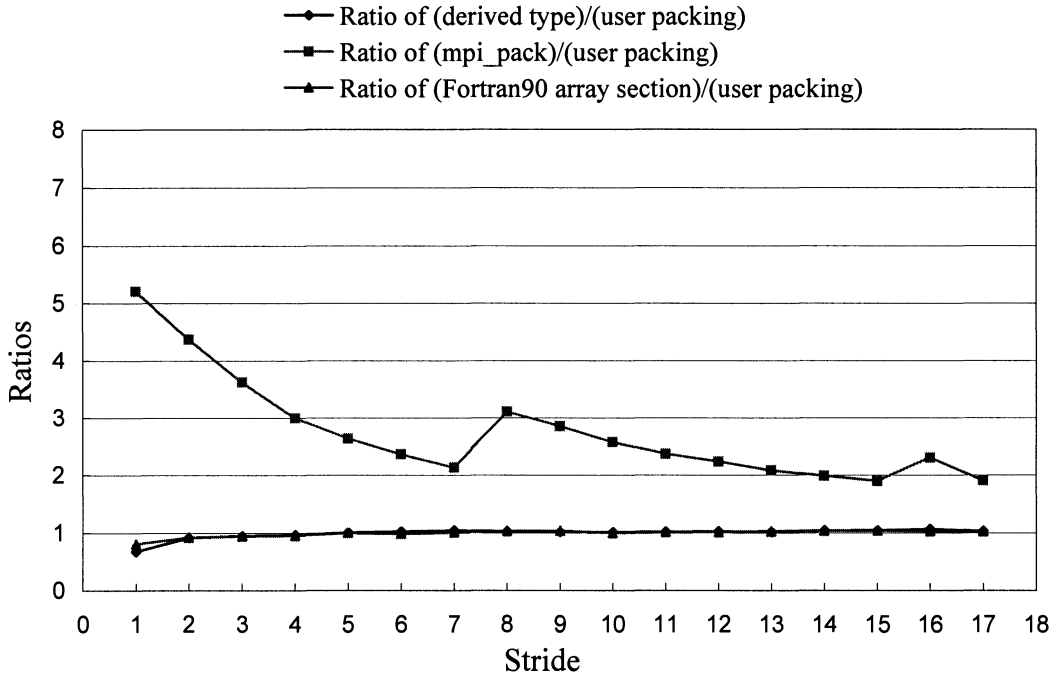

Figure 3.8 Test 2 ratios for the Intel/Myrinet cluster within a node

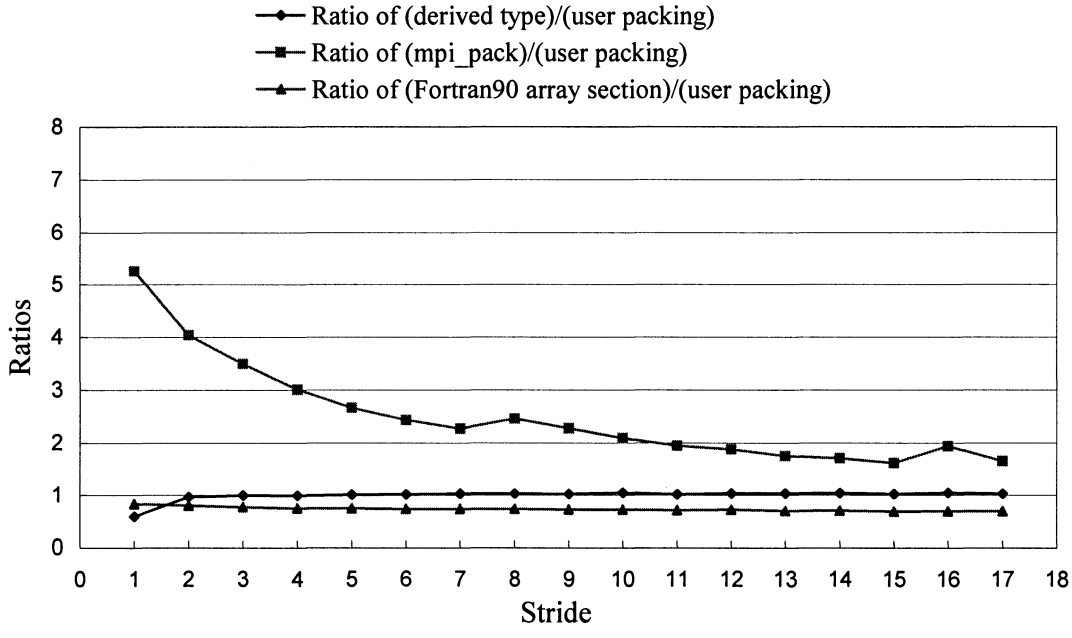

Figure 3.9 Test 2 ratios for the Intel/Myrinet cluster between nodes

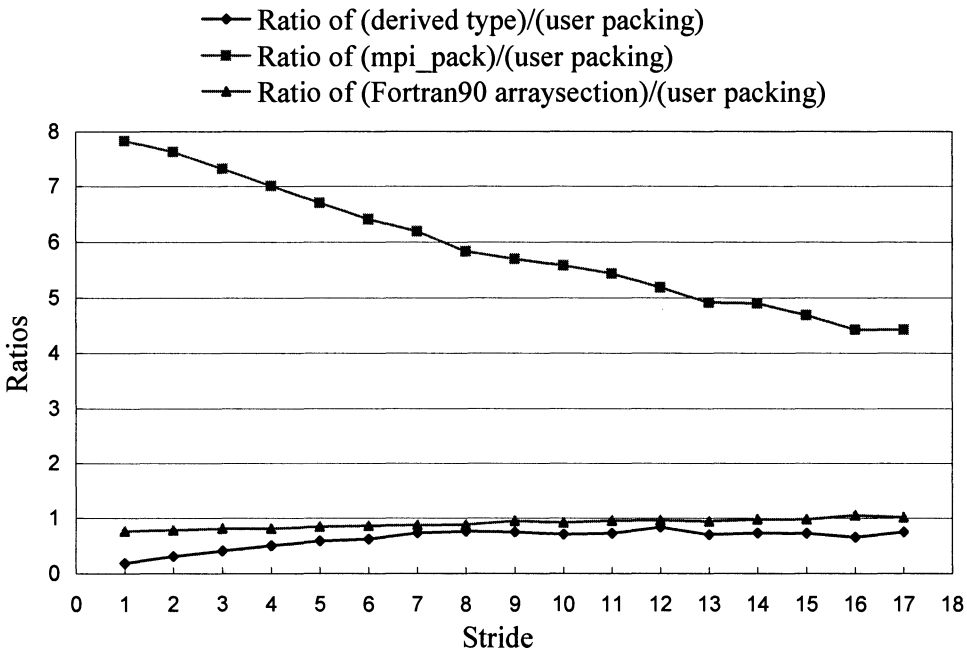

Figure 3.10 Test 2 ratios for the IBM DataStar within a node

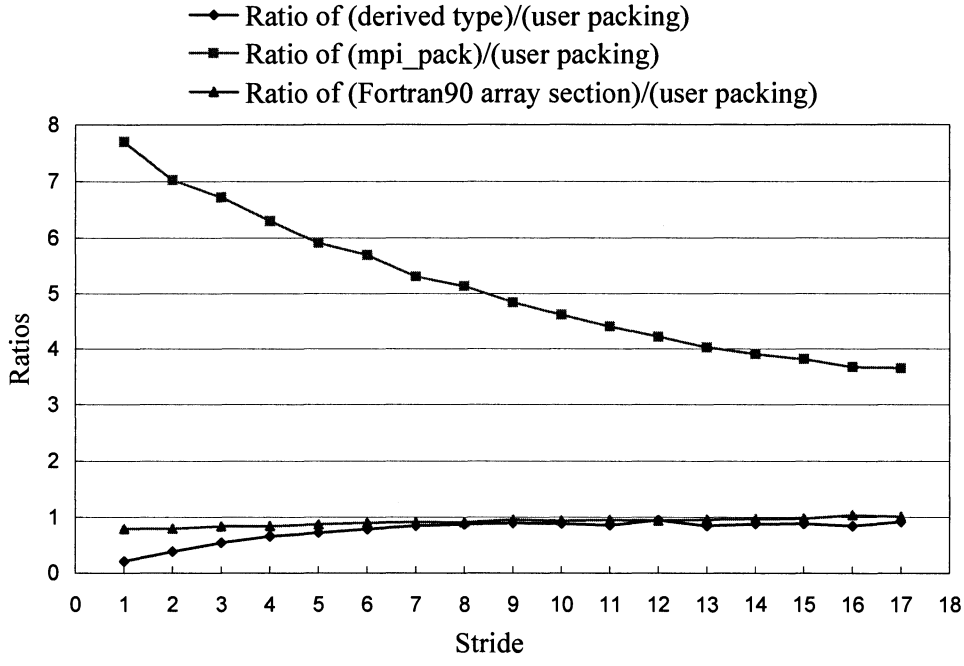

Figure 3.11 Test 2 ratios for the IBM DataStar between nodes

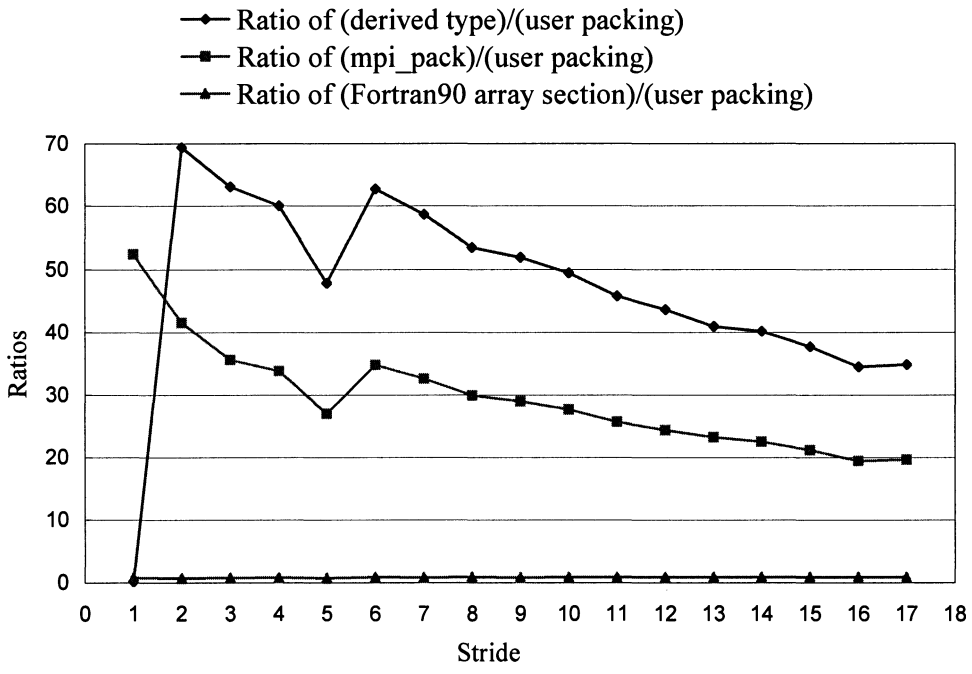

Figure 3.12 Test 2 ratios for the Cray X1 within a node

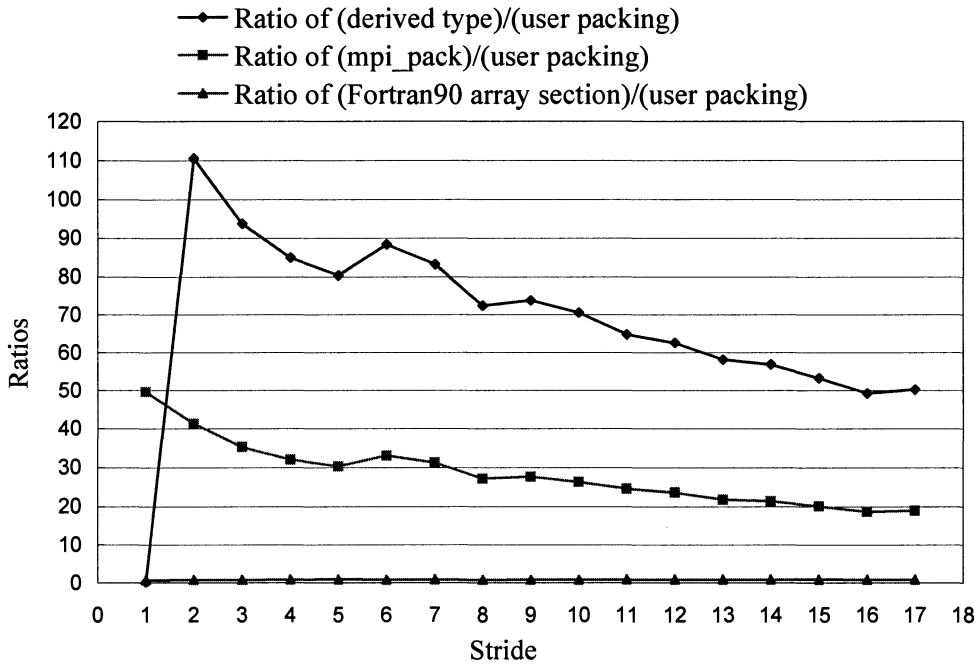

Figure 3.13 Test 2 ratios for the Cray X1 between nodes

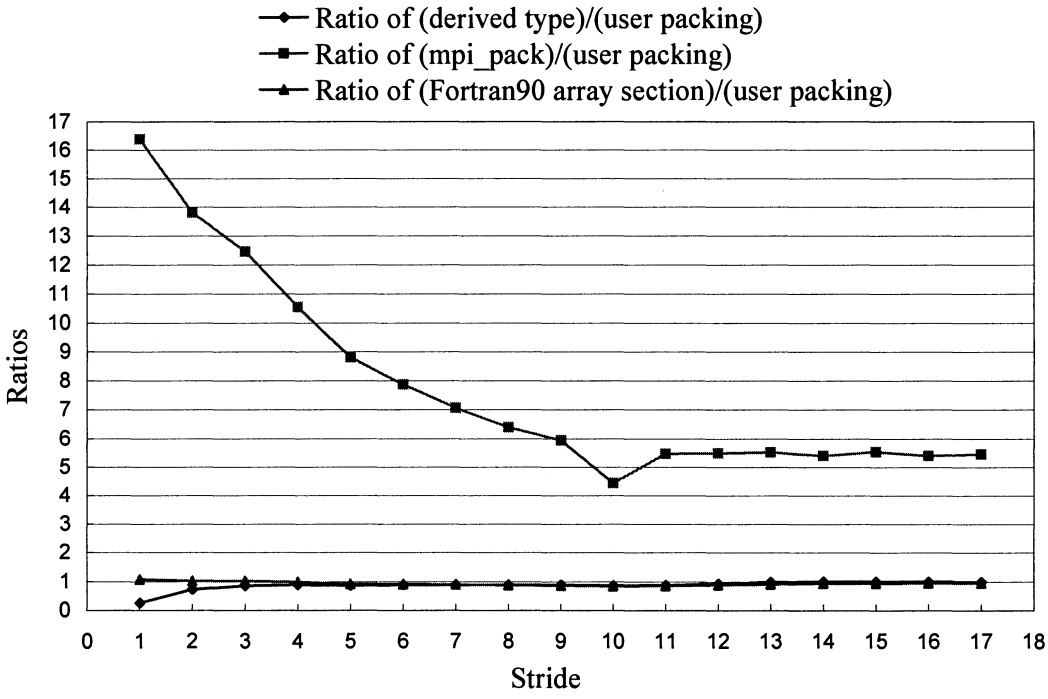

Figure 3.14 Test 2 ratios for the Cray XT3 between nodes

3.3 Test 3. Sending the lower triangular portion of a 2-dimensional array

Sometimes one would like to send only the lower (or upper) triangular portion of a 2-dimensional array instead of sending the entire array. Since this data is not contiguous in memory, it can be sent using MPI derived types, mpi_pack and mpi_unpack, or the user can pack the data into a contiguous temporary array. Let A be a double precision array of dimension n by n. To compare these three methods, we chose n to be 16, 32, 64, 128, 200, 400, 600, and 800. The timing programs for these methods are similar to the descriptions in section 3.1.

The MPI derived type used to send this data is called ltriangle and is defined as follows:

```
do i = 1, n                ! initialize the block and disp arrays
    block(i) = n+1-i
    disp(i) = (i-1)*(n+1)
enddo
call mpi_type_indexed (n, block, disp, mpi_double_precision, &
                       ltriangle, ierror)
call mpi_type_commit (ltriangle, ierror)
```

For the mpi_pack method, the data was packed into temp1 as follows:

```
position = 0
do i = 1, n
    call mpi_pack(A(i, i), n-i+1, dp, temp1, size, position, comm, ierror)
enddo
```

The user packing method packs the non-contiguous data into the temporary array temp1 of length n*(n+1)/2 as follows:

```
index = 1
do j = 1, n
    do i = j, n
        temp1(index) = A(i, j)
        index = index+1
    enddo
enddo
```

Performance results for test 3 are presented in Figures 3.15 through 3.21 and include stride one results. For each machine, the performance ratios within a node and between nodes are similar. For the Cray XT3 and the IBM machines, MPI derived types performed the best. For the Cray X1, mpi_pack performed best and both mpi_pack and MPI derived types performed much better than user packing for the larger matrix sizes. For the Intel/Myrinet cluster, all three methods performed about the same and no single method performed best for all matrix sizes.

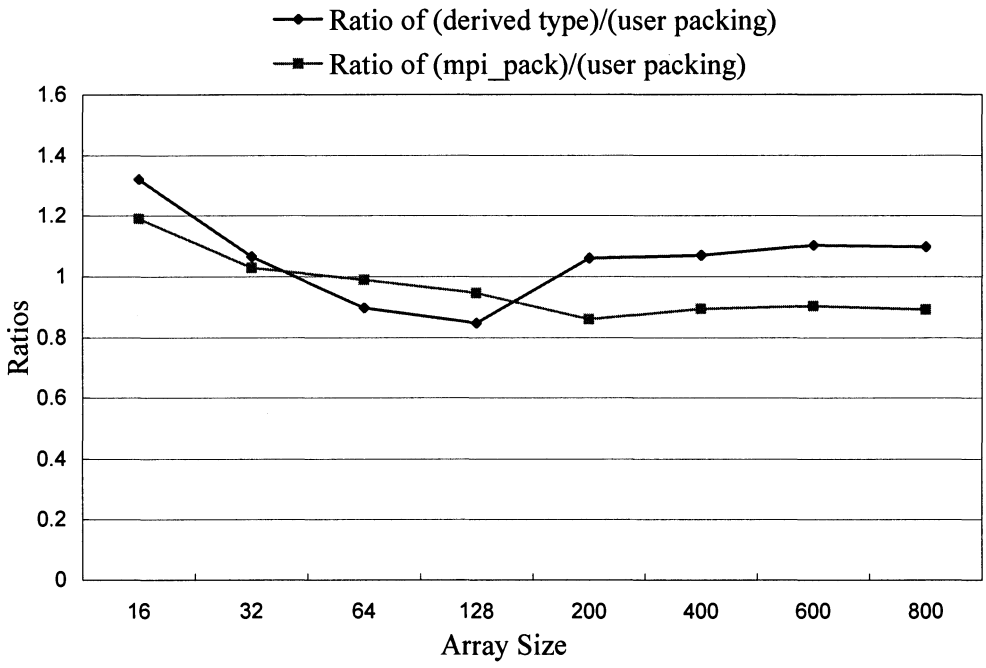

Figure 3.15 Test 3 ratios for the Intel/Myrinet cluster within a node

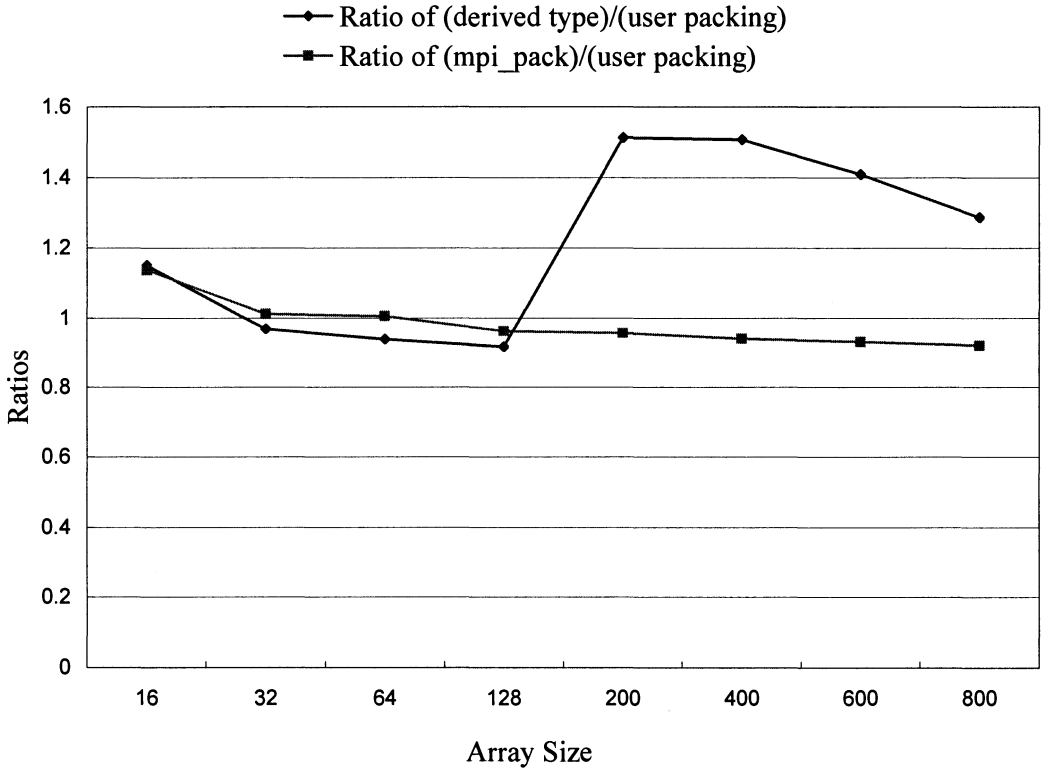

Figure 3.16 Test 3 ratios for the Intel/Myrinet cluster between nodes

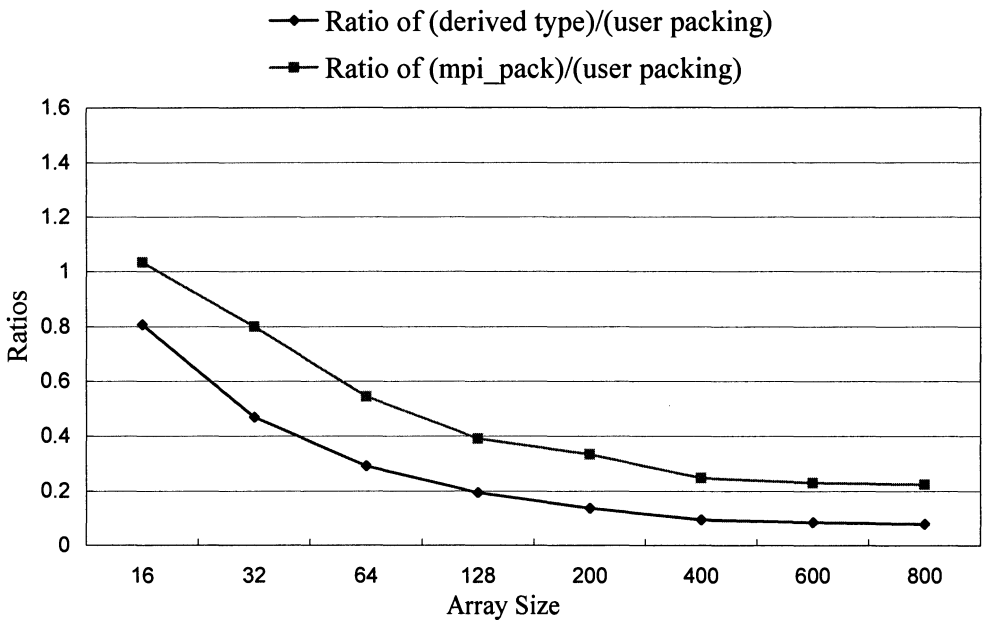

Figure 3.17 Test 3 ratios for the IBM DataStar within a node

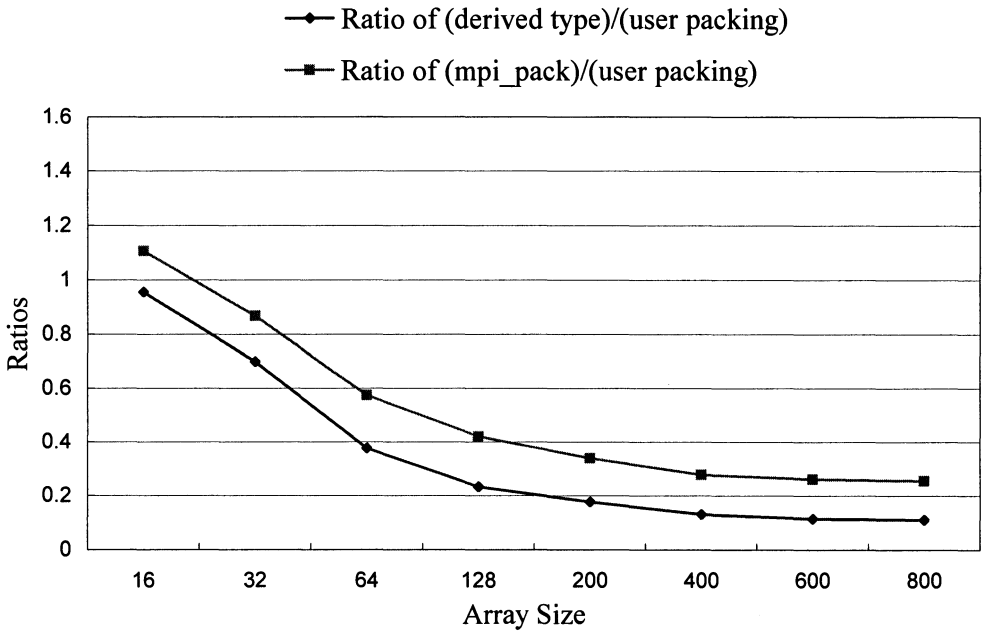

Figure 3.18 Test 3 ratios for the IBM DataStar between nodes

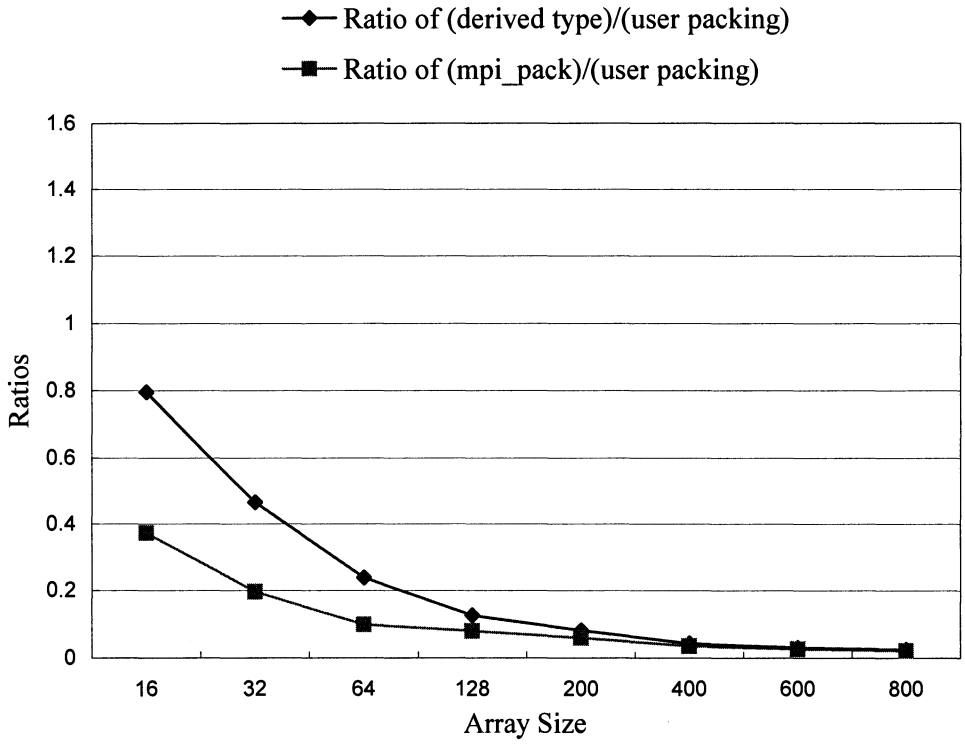

Figure 3.19 Test 3 ratios for the Cray X1 within a node

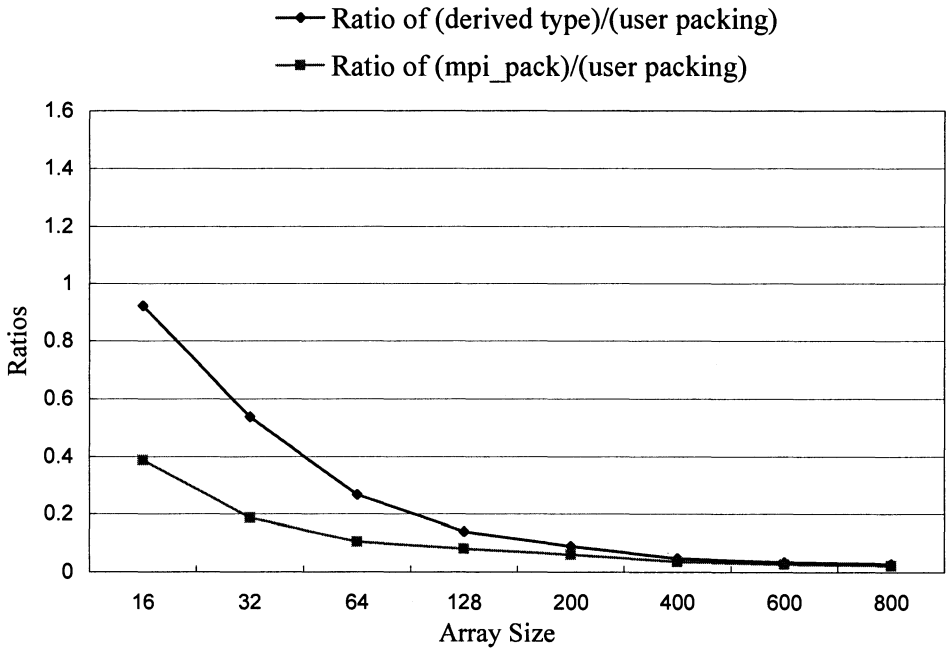

Figure 3.20 Test 3 ratios for the Cray X1 between nodes

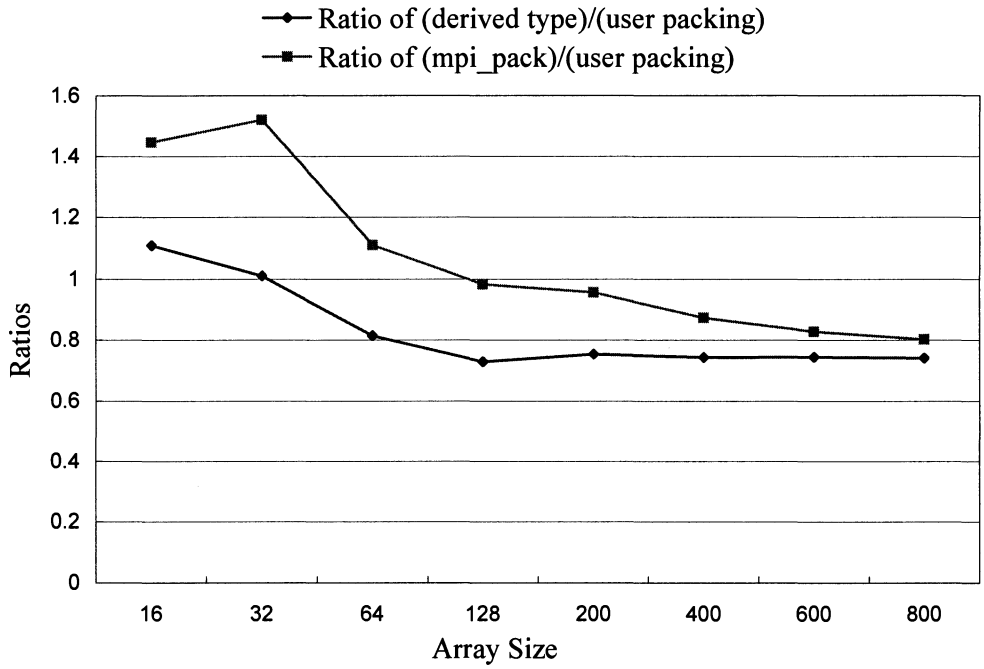

Figure 3.21 Test 3 ratios for the Cray XT3 between nodes

3.4 Test 4. Sending block diagonal of a 2-dimensional array

The blocks in block diagonal arrays can be sent using MPI derived types, mpi_pack and mpi_unpack, or the user can pack the blocks into a contiguous temporary array. Let A be a double precision array with the size of n by n with blocksize m. We chose n be 64, 128, 512 and 1024, and m be 4, 8 and 16.

The MPI derived type used to send this data is called bdiag and is defined as follows:

```
blocklen(1:n) = m
do i = 1, n
    disp(i) = (i-1)*n + INT((i-1)/m)*m
enddo
call mpi_type_indexe d (n, blocklen, disp, dp, bdiag, ierror)
call mpi_type_commit (bdiag, ierror)
```

The timing was done exactly as was done for in test 1 with the row MPI derived type replaced with bdiag.

For the mpi_pack method, the data was packed into temp1 as follows

```
position = 0
do i = 1, n
   call mpi_pack (A(i, i), m, dp, temp1, size, position, ... )
enddo
```

The user packing method packs the non-contiguous data into the temporary array temp1 of size m by n as follows:

```
count = n/m          ! the number of blocks
do i = 1, count
   j = i*m
   temp1(1:m, (j-m+1):j) = A((j-m+1):j, (j-m+1):j)
enddo
```

Figures 3.22 through 3.28 present the comparative performance data. Notice that the using MPI derived types gives the best performance for the IBM DataStar, the Cray XT3, and the Intel/Myrinet cluster (except for one the n = 1024 and m = 16). For the Cray X1, MPI derived types gives the poorest performance.

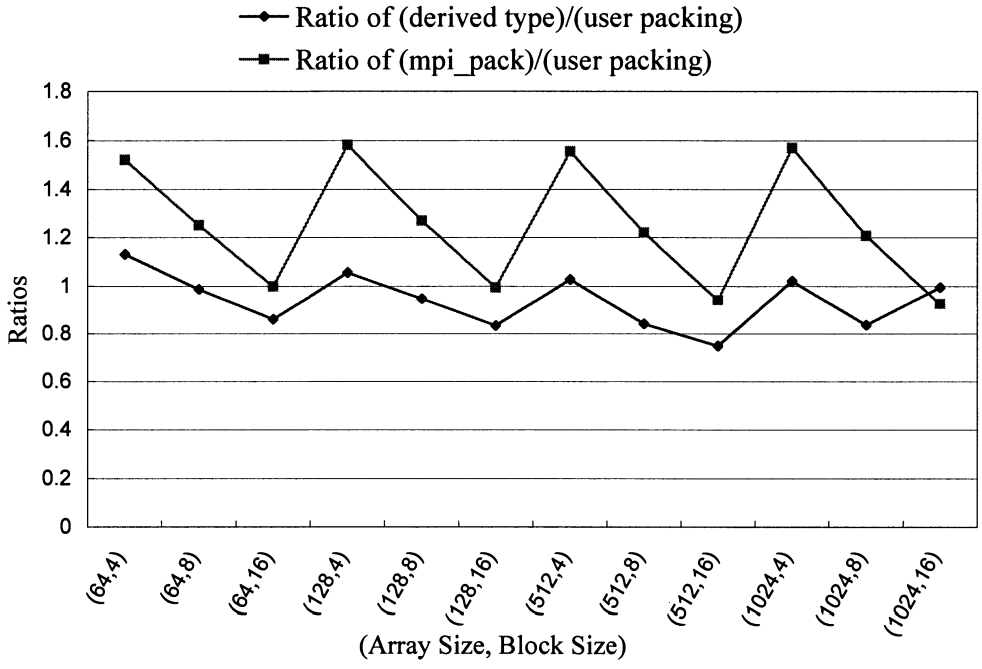

Figure 3.22 Test 4 ratios for the Intel/Myrinet cluster within a node.

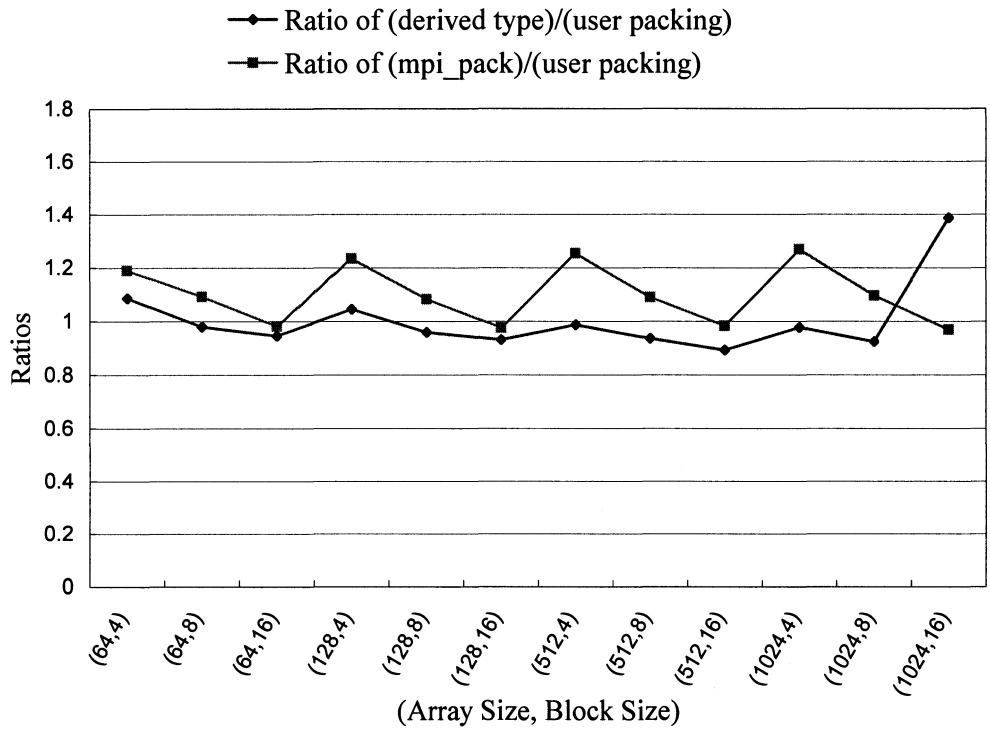

Figure 3.23 Test 4 ratios for the Intel/Myrinet cluster between nodes.

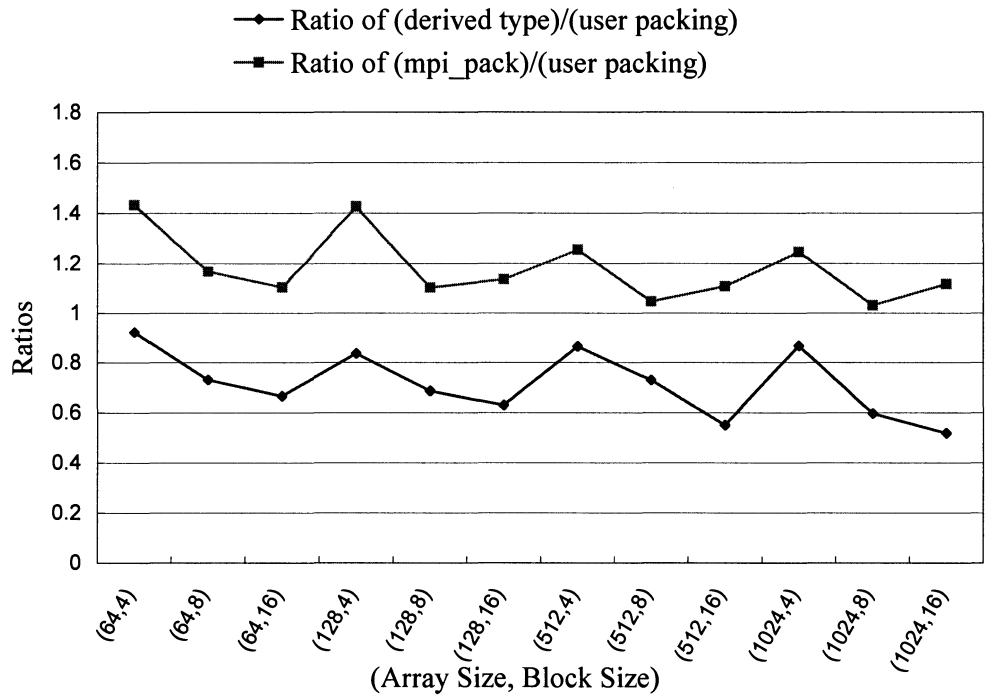

Figure 3.24 Test 4 ratios for the IBM DataStar within a node.

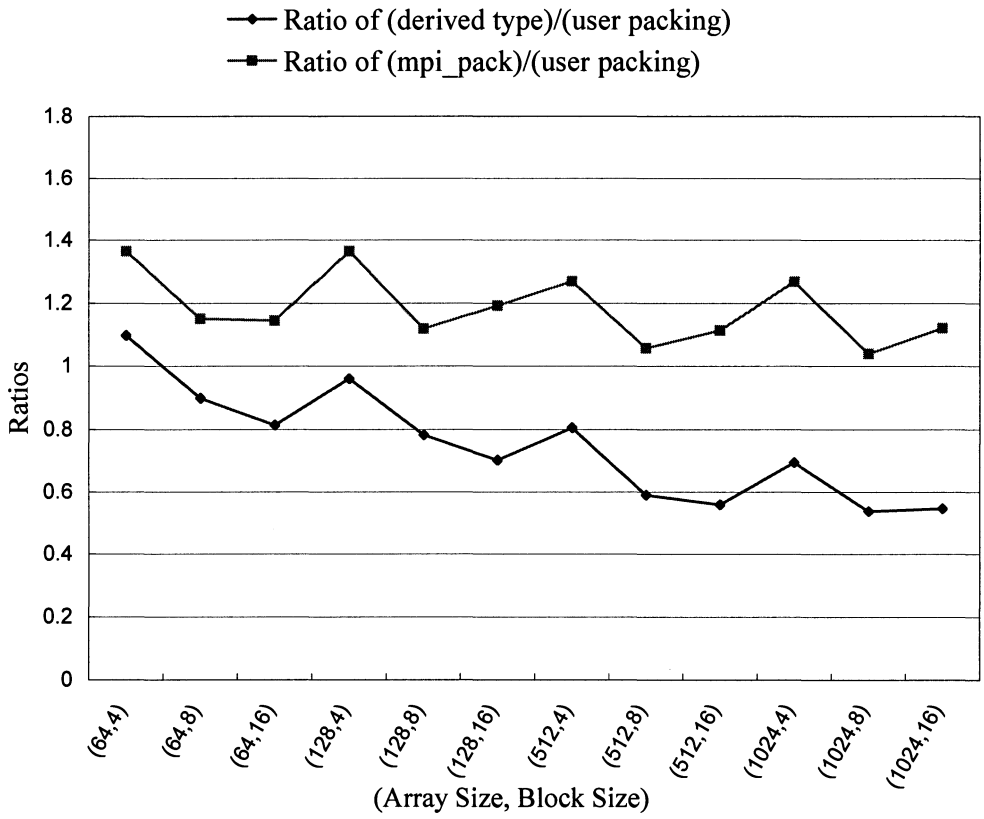

Figure 3.25 Test 4 ratios for IBM DatatStar between nodes.

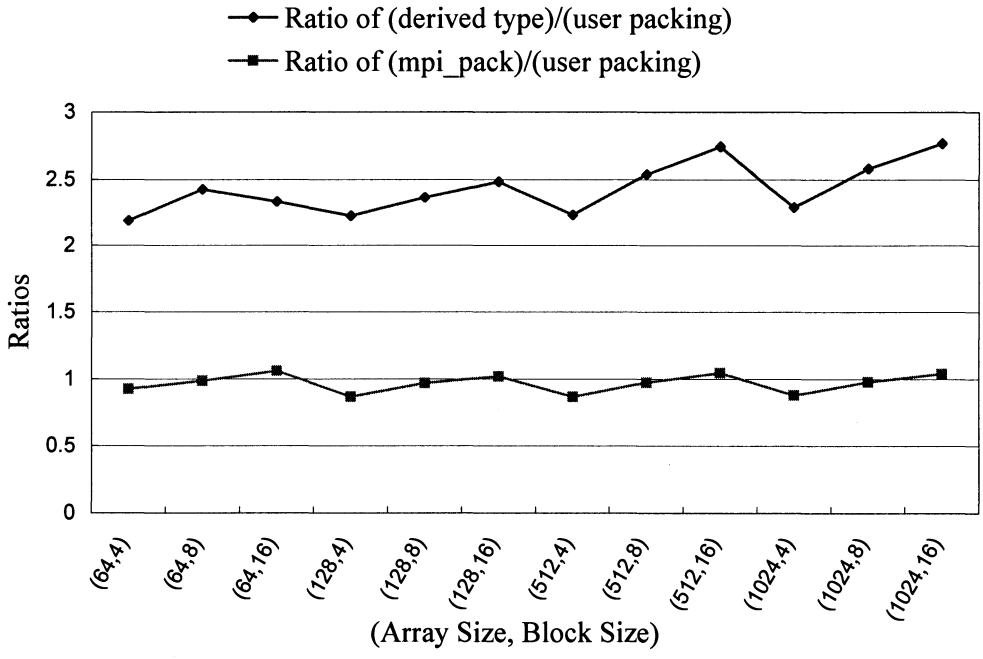

Figure 3.26 Test 4 ratios for the Cray X1 within a node.

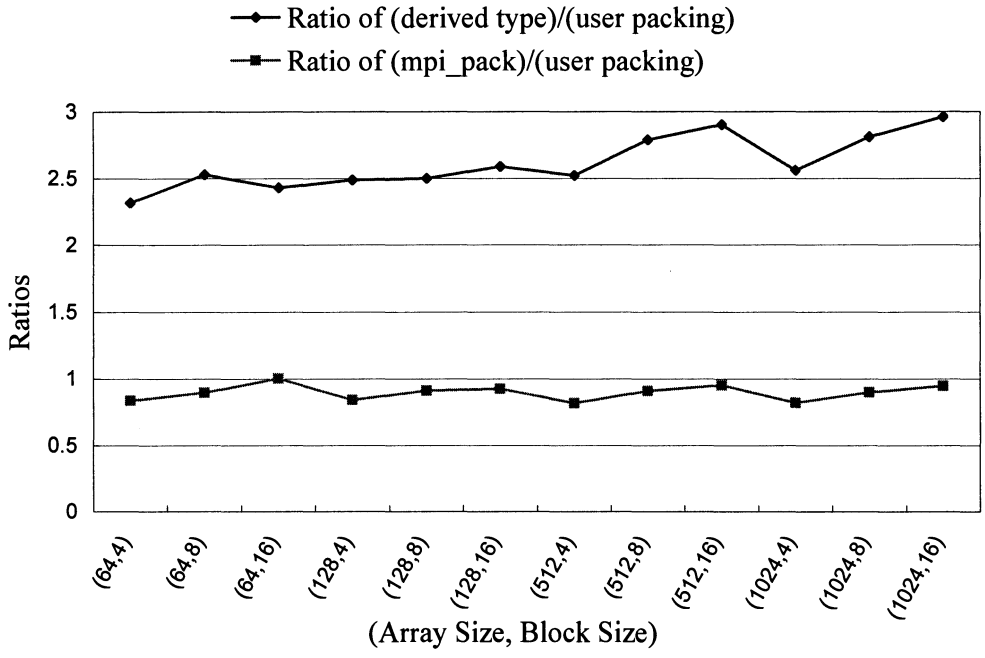

Figure 3.27 Test 4 ratios for the Cray X1 between nodes.

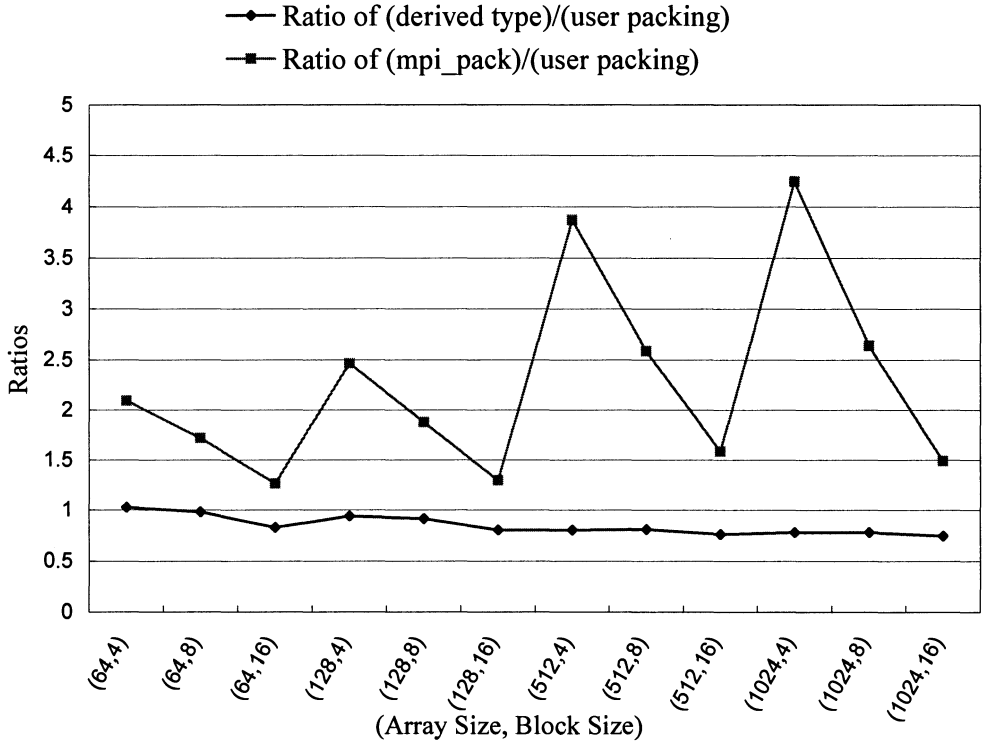

Figure 3.28 Test 4 ratios for the Cray XT3 between nodes.

4. Conclusions

The purpose of this paper is to evaluate the performance and ease-of-use of four methods of sending non-contiguous data for a variety of constructs commonly found in scientific applications. The methods of sending non-contiguous data considered in this paper are: (1) using Fortran 90 array sections, (2) using MPI derived types, (3) using explicit user packing into a contiguous buffer, and (4) using explicit packing with mpi_pack and mpi_unpack into a contiguous buffer.

We found that using both MPI derived types and Fortran 90 array sections to be easy-to-use and much easier to use than methods 3 and 4 listed above. However, Fortran 90 array sections can only be used to send noncontiguous data that can be represented as a Fortran 90 array section. In addition, Fortran 90 array sections cannot be used with nonblocking sends and receives, see [11]. Clearly, Fortran 90 array sections cannot be used in C and C++ MPI programs. However, MPI derived types can be used in Fortran, C, and C++ programs.

The performance of MPI derived types will depend on the quality of their implementation. For the IBM DataStar and the Cray XT3, MPI derived types performed best in all tests. For the Cray X1 and the Intel/Myrinet cluster, performance results were mixed with no single method always outperforming the other methods. Our results show that MPICH2's derived data type implementation is better than MPICH1's implementation. The MPI on the Intel/Myrinet cluster is based on MPICH1 and not on the newer MPICH2.

5. Acknowledgments

We would like to thank Cray for giving us access to their Cray X1 and Cray XT3 machines. We would like to thank the San Diego Supercomputer Center for giving us access to their IBM DataStar. We also would like to thank Iowa State University for access to their Intel/Myrinet cluster.

Chapter 3. General Conclusions

The purpose of this paper is to evaluate the performance and ease-of-use of four methods for sending non-contiguous data in MPI programs. The methods considered in this paper are: (1) using Fortran 90 array sections, (2) using MPI derived types, (3) using explicit user packing into a contiguous buffer, and (4) using explicit packing with mpi_pack and mpi_unpack into a contiguous buffer. Four communication tests, commonly found in scientific applications, were designed and run with a variety of message sizes on a Cray X1, a Cray XT3, an IBM Power4 system, and on an Intel/Myrinet cluster. Methods (1) and (2) were much easier to use than the other methods.

We found that using both MPI derived types and Fortran 90 array sections to be easy-to-use and the performance of MPI derived types will depend on the quality of their implementation. For the IBM DataStar and the Cray XT3, MPI derived types performed best in all tests. For the Cray X1 and the Intel/Myrinet cluster, performance results were mixed with no single method always outperforming the other methods.

Bibliography

Note: All website information are retrieved on April 13, 2005.

[1]. The Cray X1 Computer, http://www.cray.com/products/x1e/index.html .

[2]. The Cray XT3 Computer, http://www.cray.com/products/xt3/index.html

[3]. The IBM DataStar Computer at the San Diego Supercomputer Center, http://www.sdsc.edu/user_services/datastar/ .

[4]. The Intel/Myrinet Cluster at Iowa State University, http://andrew.ait.iastate.edu/HPC/hpc-class/ .

[5]. The Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard", 1995.

[6]. The Message Passing Interface Forum, "MPI-2: Extensions to the Message-Passing Interface", 1997.

[7]. Special Karlsruher MPI (SKaMPI), http://liinwww.ira.uka.de/~skampi/ .

[8]. Ralf Reussner, Jesper Larsson Träff, Gunnar Hunzelmann. "A Benchmark for MPI Derived Datatypes". In Recent Advances in Parallel Virtual Machine and Message Passing Interface. 7th European PVM/MPI Users' Group Meeting, volume 1908 of Lecture Notes in Computer Science, pages 10-17, 2000.

[9]. Glenn R. Luecke, Jing Yuan, Silvia Spanoyannis, Marina Kraeva. " Performance and Scalability of MPI on PC clusters", Concurrency and Computation: Practice and Experience, 2004: volume 16, pages 79-107.

[10]. MPICH2, http://www-unix.mcs.anl.gov/mpi/mpich2/ .

[11]. "Problems Due to Data Copying and Sequence Association" topic from MPI-2.0 document, http://www.mpi-forum.org/docs/mpi-20-html/node236.htm#Node238

[12]. Intel MPI Benchmarks, http://www.intel.com/software/products/cluster/mpi/mpi_benchmarks_lic.htm

[13]. Myrinet Inc. http://www.myri.com/

Appendix

DT: MPI Derived Type method UP: User Packing method
MP: mpi_pack/mpi_unpack method FA: Fortran 90 Array section method
DP/UP: the ratio of (derived type)/(user packing)
MP/UP: the ratio of (mpi_pack)/(user packing)
FA/UP: the ratio of (Fortran90 array section)/(user packing)

All timing results are in milliseconds.

Note: All raw data from Cray XT3 was obtained from a machine with system software not fully optimized. Cray requested that we not publish the raw data, but we can publish the ratios presented in the text of this paper.

Table 1. Test 1 sending N rows of 500 by 1000 double precision array for the Intel/Myrinet cluster within a node.

N	DT	UP	MP	FA	DT/UP	MP/UP	FA/UP
1	2.53E-01	3.12E-01	6.44E-01	3.17E-01	0.81	2.06	1.01
10	6.02E-01	7.92E-01	9.39E-01	1.07E+00	0.76	1.19	1.35
50	3.21E+00	3.23E+00	2.53E+00	4.70E+00	0.99	0.79	1.46
100	5.82E+00	6.10E+00	4.17E+00	1.07E+01	0.95	0.68	1.76
300	1.66E+01	1.78E+01	1.12E+01	3.11E+01	0.93	0.63	1.75
400	2.18E+01	2.37E+01	1.64E+01	4.19E+01	0.92	0.69	1.77
499	2.64E+01	2.83E+01	2.00E+01	4.62E+01	0.93	0.70	1.63
500	1.06E+01	2.83E+01	2.02E+01	4.67E+01	0.37	0.72	1.65

Table 2. Test 1 sending N rows of 500 by 1000 double precision array for the Intel/Myrinet cluster between nodes.

N	DT	UP	MP	FA	DT/UP	MP/UP	FA/UP
1	3.32E-01	3.73E-01	6.99E-01	2.97E-01	0.89	1.87	0.80
10	9.21E-01	1.09E+00	1.22E+00	1.20E+00	0.85	1.12	1.11
50	5.51E+00	4.17E+00	3.49E+00	5.04E+00	1.32	0.84	1.21
100	1.02E+01	7.83E+00	5.95E+00	1.28E+01	1.30	0.76	1.64
300	2.43E+01	2.25E+01	1.61E+01	3.53E+01	1.08	0.72	1.57
400	3.07E+01	3.00E+01	2.12E+01	4.65E+01	1.02	0.71	1.55
499	3.62E+01	3.41E+01	2.60E+01	5.54E+01	1.06	0.76	1.62
500	1.64E+01	3.41E+01	2.64E+01	5.52E+01	0.48	0.77	1.62

Table 3. Test 1 sending N rows of 500 by 1000 double precision array for the IBM DataStar within a node.

N	DT	UP	MP	FA	DT/UP	MP/UP	FA/UP
1	2.98E-01	3.45E-01	7.86E-01	2.83E-01	0.86	2.28	0.82
10	3.80E-01	7.84E-01	9.40E-01	7.18E-01	0.48	1.20	0.92
50	6.25E-01	2.20E+00	1.49E+00	2.19E+00	0.28	0.68	1.00
100	8.28E-01	3.86E+00	2.07E+00	3.89E+00	0.21	0.54	1.01
300	1.59E+00	1.06E+01	4.23E+00	1.07E+01	0.15	0.40	1.01
400	2.01E+00	1.42E+01	5.48E+00	1.42E+01	0.14	0.39	1.00
499	2.20E+00	1.72E+01	6.20E+00	1.72E+01	0.13	0.36	1.00
500	1.49E+00	1.72E+01	6.11E+00	1.72E+01	0.09	0.36	1.00

Table 4. Test 1 sending N rows of 500 by 1000 double precision array for the IBM DataStar between nodes.

N	DT	UP	MP	FA	DT/UP	MP/UP	FA/UP
1	3.15E-01	3.56E-01	8.02E-01	2.93E-01	0.88	2.25	0.82
10	3.92E-01	8.02E-01	9.55E-01	7.41E-01	0.49	1.19	0.92
50	7.43E-01	2.29E+00	1.58E+00	2.29E+00	0.32	0.69	1.00
100	1.05E+00	4.09E+00	2.29E+00	4.09E+00	0.26	0.56	1.00
300	2.08E+00	1.13E+01	4.94E+00	1.13E+01	0.18	0.44	1.00
400	2.60E+00	1.51E+01	6.29E+00	1.50E+01	0.17	0.42	0.99
499	3.19E+00	1.84E+01	7.57E+00	1.83E+01	0.17	0.41	1.00
500	2.64E+00	1.84E+01	7.38E+00	1.83E+01	0.14	0.40	1.00

Table 5. Test 1 sending N rows of 500 by 1000 double precision array for the Cray X1 within a node.

N	DT	UP	MP	FA	DT/UP	MP/UP	FA/UP
1	4.04E+00	3.02E+00	2.05E+00	1.60E+00	1.34	0.68	0.53
10	4.27E+00	3.05E+00	2.09E+00	1.68E+00	1.40	0.68	0.55
50	4.33E+00	3.22E+00	2.11E+00	1.83E+00	1.35	0.66	0.57
100	4.44E+00	4.97E+00	6.03E+00	3.56E+00	0.89	1.21	0.72
300	5.31E+00	1.11E+01	6.58E+00	8.61E+00	0.48	0.59	0.77
400	5.96E+00	1.47E+01	7.38E+00	1.18E+01	0.41	0.50	0.80
499	6.51E+00	1.64E+01	7.89E+00	1.32E+01	0.40	0.48	0.81
500	3.40E-01	1.65E+01	7.66E+00	1.34E+01	0.02	0.47	0.81

Table 6. Test 1 sending N rows of 500 by 1000 double precision array for the Cray X1 between nodes.

N	DT	UP	MP	FA	DT/UP	MP/UP	FA/UP
1	5.35E+00	3.04E+00	2.06E+00	1.65E+00	1.76	0.68	0.54
10	5.40E+00	3.06E+00	2.08E+00	1.67E+00	1.77	0.68	0.55
50	5.50E+00	3.25E+00	2.11E+00	1.81E+00	1.69	0.65	0.56
100	5.59E+00	4.98E+00	6.05E+00	3.55E+00	1.12	1.21	0.71
300	6.43E+00	1.11E+01	6.60E+00	8.60E+00	0.58	0.59	0.77
400	7.12E+00	1.47E+01	7.44E+00	1.18E+01	0.48	0.51	0.80
499	7.65E+00	1.64E+01	7.89E+00	1.32E+01	0.47	0.48	0.80
500	3.57E-01	1.65E+01	7.65E+00	1.34E+01	0.02	0.46	0.81

Table 8. Test 2 sending various strides elements of a double precision array with size of 5000 for the Intel/Myrinet cluster within a node.

STRIDE	DT	UP	MP	FA	DT/UP	MP/UP	FA/UP
1	1.86E-01	2.74E-01	1.42E+00	2.22E-01	0.68	5.20	0.81
2	1.51E-01	1.65E-01	7.20E-01	1.52E-01	0.92	4.37	0.93
3	1.26E-01	1.33E-01	4.81E-01	1.26E-01	0.95	3.62	0.95
4	1.17E-01	1.22E-01	3.65E-01	1.16E-01	0.96	2.99	0.96
5	1.12E-01	1.11E-01	2.92E-01	1.11E-01	1.01	2.64	1.00
6	1.07E-01	1.05E-01	2.47E-01	1.03E-01	1.02	2.36	0.99
7	1.05E-01	1.01E-01	2.15E-01	1.02E-01	1.04	2.13	1.01
8	9.52E-02	9.26E-02	2.88E-01	9.59E-02	1.03	3.10	1.04
9	9.20E-02	9.02E-02	2.57E-01	9.36E-02	1.02	2.85	1.04
10	9.11E-02	9.00E-02	2.32E-01	9.00E-02	1.01	2.57	1.00
11	9.12E-02	8.94E-02	2.12E-01	9.08E-02	1.02	2.37	1.02
12	9.02E-02	8.79E-02	1.96E-01	8.94E-02	1.03	2.23	1.02
13	8.93E-02	8.77E-02	1.82E-01	9.02E-02	1.02	2.08	1.03
14	9.01E-02	8.63E-02	1.72E-01	8.97E-02	1.04	1.99	1.04
15	8.93E-02	8.58E-02	1.63E-01	8.96E-02	1.04	1.90	1.04
16	9.00E-02	8.45E-02	1.95E-01	8.72E-02	1.07	2.30	1.03
17	8.44E-02	8.17E-02	1.56E-01	8.38E-02	1.03	1.91	1.03

Table 9. Test 2 sending various strides elements of a double precision array with size of 5000 for the Intel/Myrinet cluster between nodes.

STRIDE	DT	UP	MP	FA	DT/UP	MP/UP	FA/UP
1	2.17E-01	3.70E-01	1.94E+00	3.06E-01	0.59	5.26	0.83
2	2.38E-01	2.47E-01	9.97E-01	1.98E-01	0.96	4.04	0.80
3	1.96E-01	1.97E-01	6.88E-01	1.52E-01	0.99	3.48	0.77
4	1.73E-01	1.75E-01	5.27E-01	1.31E-01	0.99	3.00	0.75
5	1.61E-01	1.60E-01	4.27E-01	1.19E-01	1.01	2.66	0.74
6	1.50E-01	1.48E-01	3.60E-01	1.08E-01	1.01	2.43	0.73
7	1.41E-01	1.38E-01	3.12E-01	1.01E-01	1.02	2.27	0.73
8	1.37E-01	1.34E-01	3.29E-01	9.80E-02	1.02	2.45	0.73
9	1.32E-01	1.30E-01	2.95E-01	9.35E-02	1.02	2.28	0.72
10	1.35E-01	1.30E-01	2.71E-01	9.33E-02	1.04	2.09	0.72
11	1.30E-01	1.29E-01	2.50E-01	9.12E-02	1.01	1.94	0.71
12	1.27E-01	1.23E-01	2.32E-01	8.86E-02	1.03	1.88	0.72
13	1.26E-01	1.23E-01	2.15E-01	8.55E-02	1.03	1.75	0.70
14	1.23E-01	1.18E-01	2.02E-01	8.33E-02	1.04	1.71	0.70
15	1.22E-01	1.19E-01	1.93E-01	8.12E-02	1.02	1.61	0.68
16	1.20E-01	1.15E-01	2.22E-01	7.95E-02	1.04	1.94	0.69
17	1.13E-01	1.10E-01	1.82E-01	7.63E-02	1.03	1.65	0.69

Table 10. Test 2 sending various strides elements of a double precision array with size of 5000 for the IBM DataStar within a node.

STRIDE	DT	UP	MP	FA	DT/UP	MP/UP	FA/UP
1	5.35E-02	2.84E-01	2.23E+00	2.16E-01	0.19	7.83	0.76
2	4.51E-02	1.46E-01	1.11E+00	1.13E-01	0.31	7.63	0.78
3	4.16E-02	1.02E-01	7.44E-01	8.23E-02	0.41	7.32	0.81
4	4.00E-02	8.05E-02	5.64E-01	6.51E-02	0.50	7.00	0.81
5	3.97E-02	6.80E-02	4.55E-01	5.71E-02	0.58	6.70	0.84
6	3.69E-02	5.99E-02	3.84E-01	5.14E-02	0.62	6.41	0.86
7	3.88E-02	5.34E-02	3.30E-01	4.65E-02	0.73	6.19	0.87
8	3.77E-02	4.98E-02	2.91E-01	4.38E-02	0.76	5.83	0.88
9	3.38E-02	4.55E-02	2.59E-01	4.30E-02	0.74	5.70	0.94
10	3.02E-02	4.31E-02	2.41E-01	3.96E-02	0.70	5.58	0.92
11	2.92E-02	4.04E-02	2.20E-01	3.85E-02	0.72	5.43	0.95
12	3.25E-02	3.90E-02	2.02E-01	3.75E-02	0.83	5.18	0.96
13	2.64E-02	3.80E-02	1.87E-01	3.58E-02	0.70	4.91	0.94
14	2.61E-02	3.60E-02	1.76E-01	3.51E-02	0.72	4.90	0.97
15	2.52E-02	3.51E-02	1.64E-01	3.42E-02	0.72	4.68	0.97
16	2.46E-02	3.76E-02	1.67E-01	3.95E-02	0.65	4.43	1.05
17	2.49E-02	3.34E-02	1.48E-01	3.39E-02	0.75	4.42	1.01

Table 11. Test 2 sending various strides elements of a double precision array with size of 5000 for the IBM DataStar between nodes.

STRIDE	DT	UP	MP	FA	DT/UP	MP/UP	FA/UP
1	6.03E-02	2.90E-01	2.23E+00	2.29E-01	0.21	7.70	0.79
2	6.16E-02	1.62E-01	1.14E+00	1.28E-01	0.38	7.02	0.79
3	6.18E-02	1.14E-01	7.66E-01	9.54E-02	0.54	6.71	0.84
4	6.05E-02	9.25E-02	5.82E-01	7.75E-02	0.65	6.30	0.84
5	5.69E-02	7.91E-02	4.68E-01	6.87E-02	0.72	5.91	0.87
6	5.43E-02	6.92E-02	3.95E-01	6.24E-02	0.78	5.70	0.90
7	5.39E-02	6.39E-02	3.39E-01	5.82E-02	0.84	5.31	0.91
8	5.19E-02	5.96E-02	3.06E-01	5.40E-02	0.87	5.13	0.91
9	4.98E-02	5.59E-02	2.70E-01	5.34E-02	0.89	4.84	0.95
10	4.73E-02	5.36E-02	2.47E-01	5.00E-02	0.88	4.61	0.93
11	4.41E-02	5.15E-02	2.27E-01	4.90E-02	0.86	4.40	0.95
12	4.75E-02	5.03E-02	2.12E-01	4.71E-02	0.95	4.21	0.94
13	4.16E-02	4.94E-02	1.99E-01	4.73E-02	0.84	4.02	0.96
14	4.16E-02	4.73E-02	1.85E-01	4.58E-02	0.88	3.90	0.97
15	4.05E-02	4.59E-02	1.75E-01	4.48E-02	0.88	3.81	0.97
16	4.09E-02	4.86E-02	1.78E-01	5.04E-02	0.84	3.67	1.04
17	4.04E-02	4.41E-02	1.61E-01	4.43E-02	0.92	3.65	1.00

Table 12. Test 2 sending various strides elements of a double precision array with size of 5000 for the Cray X1 within a node.

STRIDE	DT	UP	MP	FA	DT/UP	MP/UP	FA/UP
1	5.30E-02	2.43E-01	1.27E+01	1.84E-01	0.22	52.39	0.76
2	1.11E+01	1.60E-01	6.64E+00	1.17E-01	69.36	41.54	0.73
3	7.57E+00	1.20E-01	4.27E+00	9.47E-02	63.05	35.58	0.79
4	5.74E+00	9.56E-02	3.23E+00	7.73E-02	60.04	33.78	0.81
5	4.59E+00	9.59E-02	2.59E+00	6.84E-02	47.85	26.99	0.71
6	3.86E+00	6.16E-02	2.14E+00	5.25E-02	62.69	34.79	0.85
7	3.38E+00	5.77E-02	1.88E+00	4.89E-02	58.66	32.56	0.85
8	2.89E+00	5.40E-02	1.61E+00	4.60E-02	53.46	29.86	0.85
9	2.56E+00	4.94E-02	1.43E+00	4.17E-02	51.89	28.94	0.84
10	2.32E+00	4.69E-02	1.30E+00	3.98E-02	49.42	27.66	0.85
11	2.11E+00	4.60E-02	1.18E+00	4.02E-02	45.78	25.69	0.87
12	1.94E+00	4.45E-02	1.09E+00	3.87E-02	43.61	24.40	0.87
13	1.79E+00	4.36E-02	1.01E+00	3.86E-02	40.96	23.24	0.89
14	1.67E+00	4.15E-02	9.35E-01	3.68E-02	40.18	22.54	0.89
15	1.56E+00	4.12E-02	8.74E-01	3.66E-02	37.71	21.20	0.89
16	1.46E+00	4.24E-02	8.24E-01	3.70E-02	34.45	19.42	0.87
17	1.38E+00	3.97E-02	7.79E-01	3.49E-02	34.84	19.63	0.88

Table 13. Test 2 sending various strides elements of a double precision array with size of 5000 for the Cray X1 between nodes.

STRIDE	DT	UP	MP	FA	DT/UP	MP/UP	FA/UP
1	4.60E-02	2.55E-01	1.27E+01	1.80E-01	0.18	49.62	0.70
2	1.71E+01	1.54E-01	6.37E+00	1.23E-01	110.52	41.28	0.80
3	1.14E+01	1.21E-01	4.28E+00	9.46E-02	93.75	35.24	0.78
4	8.55E+00	1.01E-01	3.23E+00	8.85E-02	84.90	32.03	0.88
5	6.84E+00	8.51E-02	2.58E+00	7.75E-02	80.38	30.29	0.91
6	5.71E+00	6.46E-02	2.14E+00	5.51E-02	88.31	33.05	0.85
7	4.90E+00	5.90E-02	1.84E+00	5.13E-02	83.19	31.26	0.87
8	4.30E+00	5.94E-02	1.61E+00	4.80E-02	72.41	27.14	0.81
9	3.83E+00	5.19E-02	1.43E+00	4.30E-02	73.80	27.61	0.83
10	3.47E+00	4.92E-02	1.29E+00	4.11E-02	70.55	26.25	0.84
11	3.12E+00	4.82E-02	1.18E+00	4.22E-02	64.74	24.51	0.88
12	2.88E+00	4.60E-02	1.08E+00	4.03E-02	62.56	23.57	0.88
13	2.65E+00	4.55E-02	9.90E-01	4.02E-02	58.17	21.76	0.88
14	2.47E+00	4.35E-02	9.29E-01	3.84E-02	56.85	21.38	0.88
15	2.32E+00	4.36E-02	8.74E-01	3.97E-02	53.30	20.07	0.91
16	2.16E+00	4.38E-02	8.18E-01	3.83E-02	49.24	18.66	0.87
17	2.04E+00	4.07E-02	7.71E-01	3.62E-02	50.22	18.96	0.89

Table 15. Test 3 sending the lower triangular portion of a 2-dimension double precision array for various sizes for the Intel/Myrinet cluster within a node.

Array Size	DT	UP	MP	DT/UP	MP/UP
16	2.53E-02	1.92E-02	2.28E-02	1.32	1.19
32	4.92E-02	4.62E-02	4.76E-02	1.06	1.03
64	1.24E-01	1.38E-01	1.36E-01	0.90	0.99
128	4.07E-01	4.80E-01	4.54E-01	0.85	0.95
200	1.26E+00	1.19E+00	1.02E+00	1.06	0.86
400	4.29E+00	4.01E+00	3.59E+00	1.07	0.89
600	9.43E+00	8.56E+00	7.74E+00	1.10	0.90
800	1.66E+01	1.51E+01	1.35E+01	1.10	0.89

Table 16. Test 3 sending the lower triangular portion of a 2-dimension double precision array for various sizes for the Intel/Myrinet cluster between nodes.

Array Size	DT	UP	MP	DT/UP	MP/UP
16	3.37E-02	2.93E-02	3.33E-02	1.15	1.14
32	6.71E-02	6.92E-02	7.01E-02	0.97	1.01
64	1.87E-01	1.99E-01	2.00E-01	0.94	1.01

128	5.80E-01	6.34E-01	6.09E-01	0.92	0.96
200	2.11E+00	1.39E+00	1.33E+00	1.51	0.96
400	7.43E+00	4.93E+00	4.63E+00	1.51	0.94
600	1.51E+01	1.07E+01	9.95E+00	1.41	0.93
800	2.41E+01	1.87E+01	1.72E+01	1.29	0.92

Table 17. Test 3 sending the lower triangular portion of a 2-dimension double precision array for various sizes for the IBM DataStar within a node.

Array Size	DT	UP	MP	DT/UP	MP/UP
16	1.71E-02	2.12E-02	2.19E-02	0.81	1.03
32	2.24E-02	4.78E-02	3.82E-02	0.47	0.80
64	4.43E-02	1.52E-01	8.27E-02	0.29	0.54
128	1.08E-01	5.55E-01	2.17E-01	0.19	0.39
200	1.81E-01	1.32E+00	4.41E-01	0.14	0.33
400	4.68E-01	4.94E+00	1.23E+00	0.09	0.25
600	9.33E-01	1.10E+01	2.54E+00	0.08	0.23
800	1.56E+00	1.97E+01	4.44E+00	0.08	0.23

Table 18. Test 3 sending the lower triangular portion of a 2-dimension double precision array for various sizes for the IBM DataStar between nodes.

Array Size	DT	UP	MP	DT/UP	MP/UP
16	2.78E-02	2.91E-02	3.22E-02	0.96	1.11
32	4.29E-02	6.14E-02	5.32E-02	0.70	0.87
64	6.43E-02	1.71E-01	9.82E-02	0.38	0.57
128	1.36E-01	5.86E-01	2.46E-01	0.23	0.42
200	2.37E-01	1.34E+00	4.55E-01	0.18	0.34
400	6.65E-01	5.11E+00	1.42E+00	0.13	0.28
600	1.30E+00	1.14E+01	2.97E+00	0.11	0.26
800	2.25E+00	2.03E+01	5.17E+00	0.11	0.25

Table 19. Test 3 sending the lower triangular portion of a 2-dimension double precision array for various sizes for the Cray X1 within a node.

Array Size	DT	UP	MP	DT/UP	MP/UP
16	1.48E-01	1.87E-01	6.96E-02	7.95E-01	3.73E-01
32	2.71E-01	5.83E-01	1.14E-01	4.65E-01	1.96E-01
64	5.23E-01	2.21E+00	2.19E-01	2.37E-01	9.94E-02
128	1.05E+00	8.35E+00	6.66E-01	1.26E-01	7.98E-02
200	1.63E+00	2.00E+01	1.19E+00	8.14E-02	5.95E-02

400	3.38E+00	7.79E+01	2.79E+00	4.34E-02	3.58E-02
600	5.29E+00	1.71E+02	4.60E+00	3.09E-02	2.69E-02
800	7.59E+00	3.01E+02	6.67E+00	2.52E-02	2.22E-02

Table 20. Test 3 sending the lower triangular portion of a 2-dimension double precision array for various sizes for the Cray X1 between nodes.

Array Size	DT	UP	MP	DT/UP	MP/UP
16	1.44E-01	1.57E-01	6.07E-02	0.92	0.39
32	2.74E-01	5.10E-01	9.48E-02	0.54	0.19
64	5.02E-01	1.89E+00	1.98E-01	0.27	0.11
128	9.97E-01	7.15E+00	5.82E-01	0.14	0.08
200	1.54E+00	1.72E+01	1.04E+00	0.09	0.06
400	3.21E+00	6.70E+01	2.46E+00	0.05	0.04
600	5.12E+00	1.47E+02	4.08E+00	0.03	0.03
800	7.37E+00	2.59E+02	5.97E+00	0.03	0.02

Table 22. Test 4 sending the block diagonal of a 2-dimension double precision array for various array and block sizes for the Intel/Myrinet cluster within a node.

Array Size	Block Size	DT	UP	MP	DT/UP	MP/UP
64	4	4.05E-02	3.58E-02	5.44E-02	1.06	1.41
64	8	5.64E-02	5.71E-02	7.15E-02	0.99	1.25
64	16	7.30E-02	8.49E-02	8.48E-02	0.84	0.99
128	4	6.48E-02	6.14E-02	9.70E-02	1.05	1.60
128	8	9.34E-02	9.86E-02	1.25E-01	0.89	1.13
128	16	1.44E-01	1.73E-01	1.72E-01	0.91	1.01
512	4	2.52E-01	2.45E-01	3.81E-01	0.96	1.35
512	8	2.95E-01	3.51E-01	4.28E-01	0.87	1.12
512	16	4.92E-01	6.58E-01	6.19E-01	0.79	0.96
1024	4	4.93E-01	4.83E-01	7.57E-01	0.95	1.35
1024	8	5.81E-01	6.95E-01	8.39E-01	0.83	1.11
1024	16	1.30E+00	1.30E+00	1.21E+00	0.94	0.95

Table 23. Test 4 sending the block diagonal of a 2-dimension double precision array for various array and block sizes for the Intel/Myrinet cluster between nodes.

Array Size	Block Size	DT	UP	MP	DT/UP	MP/UP
64	4	6.07E-02	5.58E-02	6.65E-02	1.03	1.29
64	8	8.03E-02	8.20E-02	8.96E-02	0.98	1.13
64	16	1.27E-01	1.34E-01	1.31E-01	0.91	0.98
128	4	9.44E-02	9.03E-02	1.11E-01	1.04	1.21

128	8	1.37E-01	1.43E-01	1.55E-01	0.92	1.08
128	16	2.33E-01	2.51E-01	2.44E-01	0.94	0.97
512	4	3.02E-01	3.06E-01	3.84E-01	0.98	1.23
512	8	4.11E-01	4.39E-01	4.78E-01	0.93	1.07
512	16	7.04E-01	7.89E-01	7.75E-01	0.90	0.98
1024	4	5.58E-01	5.71E-01	7.24E-01	0.97	1.25
1024	8	7.65E-01	8.27E-01	9.07E-01	0.90	1.02
1024	16	2.10E+00	1.52E+00	1.47E+00	1.28	0.96

Table 24. Test 4 sending the block diagonal of a 2-dimension double precision array for various array and block sizes for the IBM DataStar within a node.

Array Size	Block Size	DT	UP	MP	DT/UP	MP/UP
64	4	5.93E-02	6.44E-02	9.22E-02	0.92	1.43
64	8	6.05E-02	8.29E-02	9.67E-02	0.73	1.17
64	16	7.71E-02	1.16E-01	1.28E-01	0.66	1.10
128	4	9.64E-02	1.15E-01	1.64E-01	0.84	1.42
128	8	1.04E-01	1.52E-01	1.67E-01	0.69	1.10
128	16	1.38E-01	2.19E-01	2.48E-01	0.63	1.14
512	4	5.35E-01	6.19E-01	7.77E-01	0.86	1.26
512	8	5.51E-01	7.56E-01	7.91E-01	0.73	1.05
512	16	5.59E-01	1.02E+00	1.13E+00	0.55	1.11
1024	4	1.07E+00	1.24E+00	1.54E+00	0.87	1.25
1024	8	9.01E-01	1.51E+00	1.56E+00	0.60	1.03
1024	16	1.03E+00	2.00E+00	2.23E+00	0.52	1.12

Table 25. Test 4 sending the block diagonal of a 2-dimension double precision array for various array and block sizes for the IBM DataStar between nodes.

Array Size	Block Size	DT	UP	MP	DT/UP	MP/UP
64	4	9.33E-02	8.51E-02	1.16E-01	1.10	1.36
64	8	9.19E-02	1.02E-01	1.18E-01	0.90	1.15
64	16	1.08E-01	1.33E-01	1.52E-01	0.81	1.14
128	4	1.32E-01	1.38E-01	1.88E-01	0.96	1.36
128	8	1.32E-01	1.69E-01	1.89E-01	0.78	1.12
128	16	1.63E-01	2.32E-01	2.76E-01	0.70	1.19
512	4	5.10E-01	6.33E-01	8.03E-01	0.81	1.27
512	8	4.50E-01	7.62E-01	8.06E-01	0.59	1.06
512	16	5.60E-01	1.00E+00	1.11E+00	0.56	1.11
1024	4	8.57E-01	1.23E+00	1.56E+00	0.69	1.27
1024	8	8.08E-01	1.50E+00	1.56E+00	0.54	1.04
1024	16	1.09E+00	2.00E+00	2.24E+00	0.55	1.12

Table 26. Test 4 sending the block diagonal of a 2-dimension double precision array for various array and block sizes for the Cray X1 within a node.

Array Size	Block Size	DT	UP	MP	DT/UP	MP/UP
64	4	5.22E-01	2.38E-01	2.20E-01	2.19	0.92
64	8	5.29E-01	2.18E-01	2.15E-01	2.42	0.98
64	16	5.33E-01	2.28E-01	2.42E-01	2.33	1.06
128	4	1.02E+00	4.58E-01	3.97E-01	2.22	0.87
128	8	1.03E+00	4.34E-01	4.21E-01	2.36	0.97
128	16	1.04E+00	4.20E-01	4.28E-01	2.48	1.02
512	4	3.96E+00	1.77E+00	1.54E+00	2.23	0.87
512	8	4.01E+00	1.58E+00	1.53E+00	2.54	0.97
512	16	4.05E+00	1.48E+00	1.54E+00	2.74	1.04
1024	4	7.87E+00	3.43E+00	3.02E+00	2.29	0.88
1024	8	7.96E+00	3.08E+00	3.01E+00	2.58	0.98
1024	16	8.04E+00	2.90E+00	3.02E+00	2.77	1.04

Table 27. Test 4 sending the block diagonal of a 2-dimension double precision array for various array and block sizes for the Cray X1 between nodes.

Array Size	Block Size	DT	UP	MP	DT/UP	MP/UP
64	4	4.97E-01	2.15E-01	1.79E-01	2.32	0.84
64	8	4.99E-01	1.97E-01	1.77E-01	2.53	0.90
64	16	5.05E-01	2.08E-01	2.08E-01	2.43	1.00
128	4	9.78E-01	3.93E-01	3.30E-01	2.49	0.84
128	8	9.73E-01	3.89E-01	3.54E-01	2.50	0.91
128	16	9.76E-01	3.77E-01	3.49E-01	2.59	0.93
512	4	3.82E+00	1.51E+00	1.24E+00	2.52	0.82
512	8	3.85E+00	1.38E+00	1.25E+00	2.79	0.91
512	16	3.85E+00	1.32E+00	1.26E+00	2.90	0.95
1024	4	7.62E+00	2.97E+00	2.43E+00	2.56	0.82
1024	8	7.61E+00	2.71E+00	2.43E+00	2.81	0.90
1024	16	7.64E+00	2.58E+00	2.45E+00	2.96	0.95